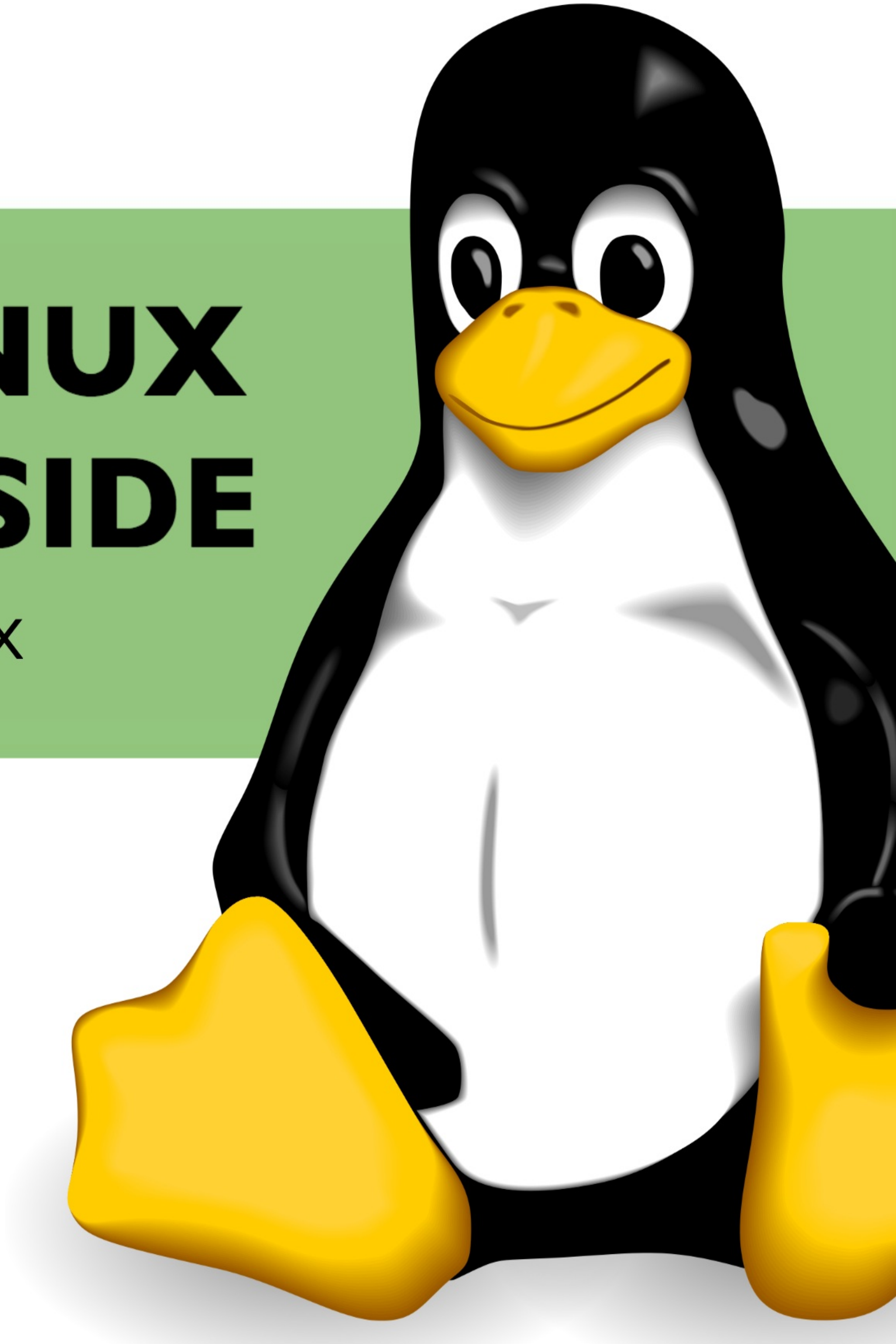
Published with GitBook

LINUX INSIDE

By 0xAX



目錄

Linux 内核揭密

chat on gitter

一系列关于 Linux 内核和其内在机理的帖子。

目的很简单 - 分享我对 Linux 内核内在机理的一点知识，帮助对 Linux 内核内在机理感兴趣的人，和其他低级话题。

问题/建议: 若有相关问题，请提交 issue。英文原文问题，找上游 repo - linux-insides 提交 issue；翻译问题，在下游 repo - linux-insides-zh 中提交 issue。

贡献条目

对于 linux-insides-zh 项目，您可以通过以下方法贡献自己的力量：

- 英文翻译，目前只提供简体中文的译文；
- 更新未被翻译的英文原本，其实就是将上游英文的更新纳入到当前项目中；
- 更新已经翻译的中文译文，其实就是查看上游英文的更新，检查是否需要对中文译文进行更新；
- 校对当前已经翻译过的中文译文，包括修改 typo，润色等工作；

翻译进度

章节	译者	翻译进度
1. Booting		正在进行
├ 1.0	@xinqiu	更新至527b2b8
├ 1.1	@hailincai	已完成
├ 1.2	@hailincai	已完成
├ 1.3	@hailincai	已完成
├ 1.4	@zmj1316	已完成
├ 1.5	@mytbk	更新至31998d14
└ 1.6	@mytbk	更新至31998d14
2. Initialization		正在进行
├ 2.0	@mudongliang	更新至44017507
├ 2.1	@dontpanic92	更新至44017507

├ 2.2	@dontpanic92	更新至44017507
├ 2.3	@dontpanic92	更新至44017507
├ 2.4	@bjwrkj	已完成
├ 2.5	@NeoCui	更新至cf32dc6c
├ 2.6	@kele1997	更新至e896e56
├ 2.7		未开始
├ 2.8		未开始
├ 2.9		未开始
└ 2.10		未开始
3. Interrupts		正在进行
├ 3.0	@littleneko	更新至57279321
├ 3.1		未开始
├ 3.2	@narcijie	正在进行
├ 3.3		未开始
├ 3.4		未开始
├ 3.5		未开始
├ 3.6		未开始
├ 3.7		未开始
├ 3.8		未开始
├ 3.9	@zhangyangjing	已完成
└ 3.10	@worldwar	已完成
4. System calls		正在进行
├ 4.0	@mudongliang	更新至194d0c83
├ 4.1	@qianmoke	已完成
├ 4.2	@qianmoke	已完成
├ 4.3		未开始
├ 4.4		未开始
├ 4.5		未开始
└ 4.6		未开始
5. Timers and time management		正在进行
├ 5.0	@mudongliang	更新至2a742fd4

├ 5.1		未开始
├ 5.2		未开始
├ 5.3		未开始
├ 5.4		未开始
├ 5.5		未开始
├ 5.6		未开始
└ 5.7		未开始
6. Synchronization primitives		正在进行
├ 6.0	@mudongliang	更新至6f85b63e
├ 6.1	@keltoy	已完成
├ 6.2	@keltoy	已完成
├ 6.3	@huxq	已完成
├ 6.4		未开始
├ 6.5		未开始
└ 6.6		未开始
7. Memory management		未开始
├ 7.0	@mudongliang	更新至f83c8ee2
├ 7.1	@choleraehyq	已完成
├ 7.2	@choleraehyq	已完成
└ 7.3		未开始
8. SMP		上游未开始
9. Concepts		正在进行
├ 9.0	@mudongliang	更新至44017507
├ 9.1	@up2wing	更新至28a39fe6
├ 9.2	@up2wing	更新至28a39fe6
├ 9.3	@up2wing	更新至28a39fe6
└ 9.4		未开始
10. DataStructures		已完成
├ 10.0	@mudongliang	更新至99138e09
├ 10.1	@oska874 @mudongliang	已完成
├ 10.2	@a1ickgu0	已完成
└ 10.3	@cposture	已完成

11. Theory		正在进行
├ 11.0	@mudongliang	更新至99ad0799
├ 11.1	@mudongliang	已完成
├ 11.2	@mudongliang	已完成
└ 11.3		未开始
12. Initial ram disk		上游未开始
13. Misc		已完成
├ 13.0	@mudongliang	更新至ddf0793f
├ 13.1	@hao-lee	更新至3ed52146
├ 13.2	@oska874	已完成
├ 13.3	@zmj1316	已完成
└ 13.4	@mudongliang	已完成
14. KernelStructures		已完成
├ 14.0	@mudongliang	更新至3cb550c0
└ 14.1	@woodpenker	更新至4521637d
15. Cgroups		正在进行
├ 15.0	@mudongliang	更新至e811ca4f
└ 15.1	@tjm-1990	更新至b420e581

翻译认领规则

为了避免多个译者同时翻译相同章节的情况出现，请按照以下规则认领自己要翻译的章节：

- 在 README.md 中查看你想翻译的章节的状态；
- 在确认想翻译的章节没有被翻译之后，开一个 issue ，告诉大家你想翻译哪一章节，同时提交申请翻译的 PR ，将 README.md 中的翻译状态修改为“正在进行”；
- 首先，从上游的英文库中得到该章节的最新版本，将修改提交到我们的中文库中；
- 然后翻译你认领的章节；
- 完成翻译之后，提交翻译内容的 PR (注：大家最好以一个文件为基本单位来提交翻译 **PR**，方便我们进行 **review**，否则可能会因为 **comments** 导致展示 **PR** 的网页变得过于冗长，不方便 **review** 修改的内容)。待 PR 被合并之后，请关闭 issue；
- 最后，将 README.md 中的翻译状态修改为“更新至上游 commit id”(注：**8** 位 **commit id**)，同时不要忘记把自己添加到 CONTRIBUTORS.md 中。

翻译前建议看 TRANSLATION_NOTES.md 。关于翻译约定，大家有任何问题或建议也请开 issue 讨论。

翻译申请回收原则

为了避免翻译申请过长时间没有任何更新，所以暂设置时间为三个月，如果三个月之内，issue 并没有任何更新，翻译申请就会被回收。

作者

@0xAX

中文维护者

@xinqiu

@mudongliang

中文贡献者

详见 CONTRIBUTORS.md

LICENSE

内核引导过程

本章介绍了Linux内核引导过程。此处你将在这看到一些描述内核加载过程的整个周期的文章：

- 从引导程序到内核 - 介绍了从启动计算机到内核执行第一条指令之前的所有阶段;
- 在内核设置代码的第一步 - 介绍了在内核设置代码的第一个步骤。你会看到堆的初始化，查询不同的参数，如 EDD，IST 等...
- 视频模式初始化和保护模式切换 - 介绍了内核设置代码中的视频模式初始化，并切换到保护模式。
- 切换 64 位模式 - 介绍切换到 64 位模式的准备工作以及切换的细节。
- 内核解压缩 - 介绍了内核解压缩之前的准备工作以及直接解压缩的细节。
- 内核地址随机化 - 介绍了 Linux 内核加载地址随机化的细节。

内核引导过程. 第一部分.

从引导加载程序内核

如果看过我在这之前的文章，你就会知道我已经开始涉足底层的代码编写。我写了一些关于 Linux x86_64 汇编的文章。同时，我开始深入研究 Linux 源代码。底层是如何工作的，程序是如何在电脑上运行的，它们是如何在内存中定位的，内核是如何管理进程和内存，网络堆栈是如何在底层工作的等等，这些我都非常感兴趣。因此，我决定去写另外的一系列文章关于 **x86_64** 框架的 Linux 内核。

注意这不是官方文档，只是学习和分享知识

需要的基础知识

- 理解 C 代码
- 理解 汇编语言 代码 （AT&T 语法）

不管怎样，如果你才开始学一些，我会在这些文章中尝试去解释一些部分。好了，小的介绍结束，我们开始深入内核和底层。

我们的文章是基于 Linux 内核 3.18 版本进行的，如果后续的内核版本有任何改变，我将作出相应的更新。

神奇的电源按钮，接下来会发生什么？

尽管这是一系列关于 Linux 内核的文章，我们在第一章并不会从内核代码开始。电脑在你按下电源开关的时候，就开始工作。主板发送信号给电源，而电源收到信号后会给电脑供应合适的电量。一旦主板收到了电源备妥信号，它会尝试启动 CPU 。CPU 则复位寄存器的所有数据，并设置每个寄存器的预定值。

80386 以及后来的 CPUs 在电脑复位后，在 CPU 寄存器中定义了如下预定义数据：

```
IP          0xfff0
CS selector 0xf000
CS base     0xffff0000
```

处理器开始在实模式工作。我们需要退回一点去理解在这种模式下的内存分段机制。从 8086 到现在的 Intel 64 位 CPU，所有 x86兼容处理器都支持实模式。8086 处理器有一个20位寻址总线，这意味着它可以对0到 2^20 位地址空间（ 1MB ）进行操作。不过它只有16位的寄存器，所以最大寻址空间是 2^16 即 0xffff （64 KB）。实模式使用段式内存管理 来管理整个内

存空间。所有内存被分成固定的65536字节（64 KB） 大小的小块。由于我们不能用16位寄存器寻址大于 64KB 的内存，一种替代的方法被设计出来了。一个地址包括两个部分：数据段起始地址和从该数据段起的偏移量。为了得到内存中的物理地址，我们要让数据段乘16并加上偏移量：

```
PhysicalAddress = Segment * 16 + Offset
```

举个例子，如果 `CS:IP` 是 `0x2000:0x0010`, 则对应的物理地址将会是：

```
>>> hex((0x2000 << 4) + 0x0010)
'0x20010'
```

不过如果我们使用16位2进制能表示的最大值进行寻址：`0xffff:0xffff` ，根据上面的公式，结果将会是：

```
>>> hex((0xffff << 4) + 0xffff)
'0x10ffef'
```

这超出 1MB 65519 字节。既然实模式下， CPU 只能访问 1MB 地址空间，`0x10ffef` 变成有 A20 缺陷的 `0x00ffef` 。

我们了解了实模式和在实模式下的内存寻址方式，让我们来回头继续来看复位后的寄存器值。

`CS` 寄存器包含两个部分：可视段选择器和隐含基址。 结合之前定义的 `CS` 基址和 `IP` 值，逻辑地址应该是：

```
0xffff0000:0xfff0
```

这种形式的起始地址为EIP寄存器里的值加上基址地址：

```
>>> 0xffff0000 + 0xfff0
'0xfffffff0'
```

得到的 `0xfffffff0` 是 4GB - 16 字节。 这个地方是 复位向量(Reset vector) 。 这是CPU在重置后期望执行的第一条指令的内存地址。它包含一个 jump 指令，这个指令通常指向BIOS入口点。举个例子，如果访问 coreboot 源代码，将看到：

```
    .section ".reset", "ax", %progbits
    .code16
.globl    _start
_start:
    .byte  0xe9
    .int   _start16bit - ( . + 2 )
    ...
```

上面的跳转指令（opcode - 0xe9）跳转到地址 `_start16bit - ( . + 2)` 去执行代码。`reset` 段是 `16` 字节代码段，起始于地址 `0xfffffff0` (`src/cpu/x86/16bit/reset16.ld`)，因此 CPU 复位之后，就会跳到这个地址来执行相应的代码 ：

```
SECTIONS {
    /* Trigger an error if I have an unuseable start address */
     _bogus = ASSERT(_start16bit >= 0xffff0000, "_start16bit too low. Please report.")
;
    _ROMTOP = 0xfffffff0;
    . = _ROMTOP;
    .reset . : {
        *(.reset);
        . = 15;
        BYTE(0x00);
    }
}
```

现在BIOS已经开始工作了。在初始化和检查硬件之后，需要寻找到一个可引导设备。可引导设备列表存储在在 BIOS 配置中, BIOS 将根据其中配置的顺序，尝试从不同的设备上寻找引导程序。对于硬盘，BIOS 将尝试寻找引导扇区。如果在硬盘上存在一个MBR分区，那么引导扇区储存在第一个扇区(512字节)的头446字节，引导扇区的最后必须是 `0x55` 和 `0xaa` ，这2个字节称为魔术字节（Magic Bytes)，如果 BIOS 看到这2个字节，就知道这个设备是一个可引导设备。举个例子：

```
;
; Note: this example is written in Intel Assembly syntax
;
[BITS 16]
[ORG  0x7c00]

boot:
    mov al, '!'
    mov ah, 0x0e
    mov bh, 0x00
    mov bl, 0x07

    int 0x10
    jmp $

times 510-($-$$) db 0

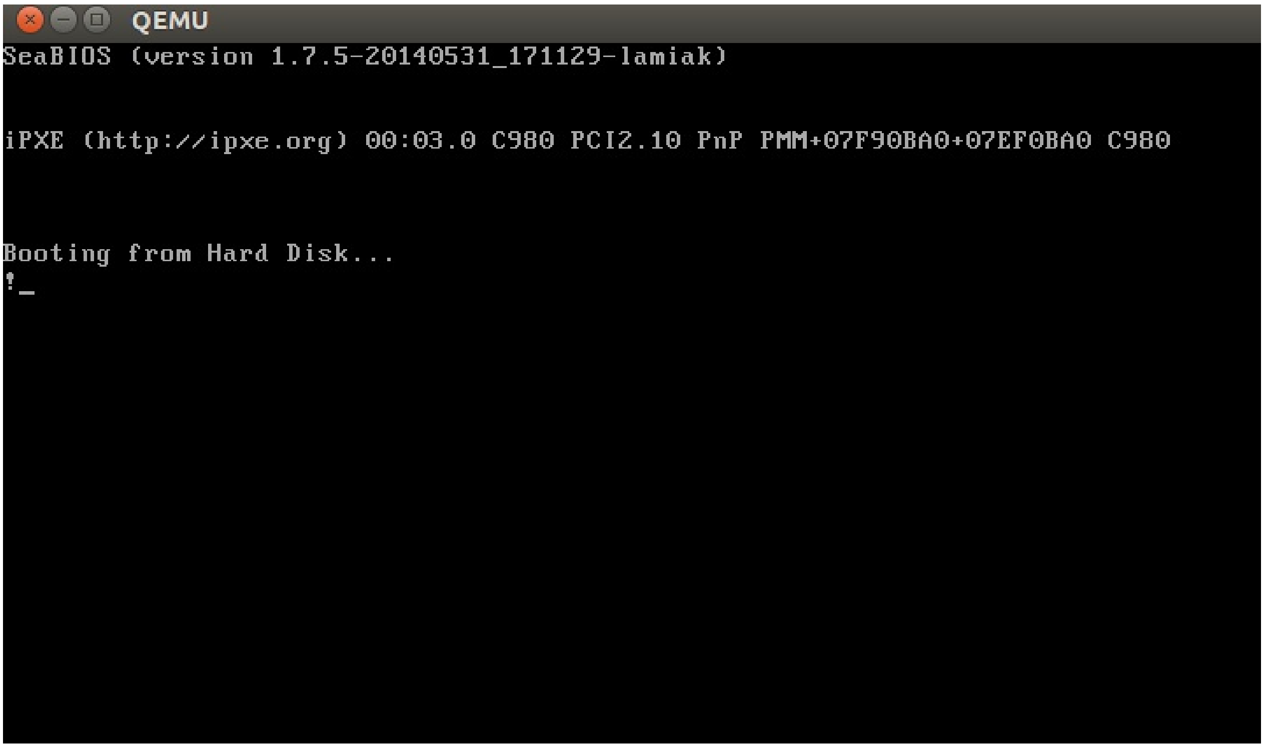
db 0x55
db 0xaa
```

构建并运行：

```
nasm -f bin boot.nasm && qemu-system-x86_64 boot
```

这让 QEMU 使用刚才新建的 `boot` 二进制文件作为磁盘镜像。由于这个二进制文件是由上述汇编语言产生，它满足引导扇区(起始设为 `0x7c00`，用Magic Bytes结束)的需求。QEMU将这个二进制文件作为磁盘镜像的主引导记录(MBR)。

将看到:



在这个例子中，这段代码被执行在16位的实模式，起始于内存0x7c00。之后调用 0x10 中断打印 `!` 符号。用0填充剩余的510字节并用两个Magic Bytes `0xaa` 和 `0x55` 结束。

可以使用 `objdump` 工具来查看转储信息：

```
nasm -f bin boot.nasm
objdump -D -b binary -mi386 -Maddr16,data16,intel boot
```

一个真实的启动扇区包含了分区表，以及用来启动系统的指令，而不是像我们上面的程序，只是输出了一个感叹号就结束了。从启动扇区的代码被执行开始，BIOS 就将系统的控制权转移给了引导程序，让我们继续往下看看引导程序都做了些什么。

NOTE: 强调一点，上面的引导程序是运行在实模式下的，因此 CPU 是使用下面的公式进行物理地址的计算的：

```
PhysicalAddress = Segment * 16 + Offset
```

而且正如我前面所说的，在实模式下，CPU 只能使用16位的通用寄存器。16位寄存器能够表达的最大数值是： `0xffff` ，所以按照上面的公式计算出的最大物理地址是：

```
>>> hex((0xffff * 16) + 0xffff)
'0x10ffef'
```

这个地址在 8086 处理器下，将被转换成地址 `0x0ffef`，原因是因为，8086 cpu 只有20位地址线，只能表示 `2^20 = 1MB` 的地址，而上面这个地址已经超出了 1MB 地址的范围，所以 CPU 就舍弃了最高位。

实模式下的 1MB 地址空间分配表：

```
0x00000000 - 0x000003FF - Real Mode Interrupt Vector Table
0x00000400 - 0x000004FF - BIOS Data Area
0x00000500 - 0x00007BFF - Unused
0x00007C00 - 0x00007DFF - Our Bootloader
0x00007E00 - 0x0009FFFF - Unused
0x000A0000 - 0x000BFFFF - Video RAM (VRAM) Memory
0x000B0000 - 0x000B7777 - Monochrome Video Memory
0x000B8000 - 0x000BFFFF - Color Video Memory
0x000C0000 - 0x000C7FFF - Video ROM BIOS
0x000C8000 - 0x000EFFFF - BIOS Shadow Area
0x000F0000 - 0x000FFFFF - System BIOS
```

如果你的记性不错，在看到这张表的时候，一定会跳出来一个问题。在上面的章节中，我说了 CPU 执行的第一条指令是在地址 `0xFFFFFFF0` 处，这个地址远远大于 `0xFFFFF`（1MB）。那么实模式下的 CPU 是如何访问到这个地址的呢？文档 coreboot 给出了答案:

```
0xFFFE_0000 - 0xFFFF_FFFF: 128 kilobyte ROM mapped into address space
```

`0xFFFFFFF0` 这个地址被映射到了 ROM，因此 CPU 执行的第一条指令来自于 ROM，而不是 RAM。

引导程序

在现实世界中，要启动 Linux 系统，有多种引导程序可以选择。比如 GRUB 2 和 syslinux。Linux内核通过 Boot protocol 来定义应该如何实现引导程序。在这里我们将只介绍 GRUB 2。

现在 BIOS 已经选择了一个启动设备，并且将控制权转移给了启动扇区中的代码，在我们的例子中，启动扇区代码是 boot.img。因为这段代码只能占用一个扇区，因此非常简单，只做一些必要的初始化，然后就跳转到 GRUB 2's core image 去执行。 Core image 的代码请参考 diskboot.img，一般来说 core image 在磁盘上存储在启动扇区之后到第一个可用分区之前。core image 的初始化代码会把整个 core image （包括 GRUB 2的内核代码和文件系统驱动）引导到内存中。 引导完成之后，grub_main将被调用。

`grub_main` 初始化控制台，计算模块基地址，设置 root 设备，读取 grub 配置文件，加载模块。最后，将 GRUB 置于 normal 模式，在这个模式中，`grub_normal_execute` (from `grub-core/normal/main.c`) 将被调用以完成最后的准备工作，然后显示一个菜单列出所用可用的操

作系统。当某个操作系统被选择之后， `grub_menu_execute_entry` 开始执行，它将调用 GRUB 的 `boot` 命令，来引导被选中的操作系统。

就像 kernel boot protocol 所描述的，引导程序必须填充 kernel setup header （位于 kernel setup code 偏移 `0x01f1` 处） 的必要字段。kernel setup header的定义开始于 arch/x86/boot/header.S：

```
    .globl hdr
hdr:
    setup_sects: .byte 0
    root_flags:  .word ROOT_RDONLY
    syssize:     .long 0
    ram_size:    .word 0
    vid_mode:    .word SVGA_MODE
    root_dev:    .word 0
    boot_flag:   .word 0xAA55
```

bootloader必须填充在 Linux boot protocol 中标记为 `write` 的头信息，比如 type_of_loader，这些头信息可能来自命令行，或者通过计算得到。在这里我们不会详细介绍所有的 kernel setup header，我们将在需要的时候逐个介绍。不过，你可以自己通过 boot protocol 来了解这些设置。

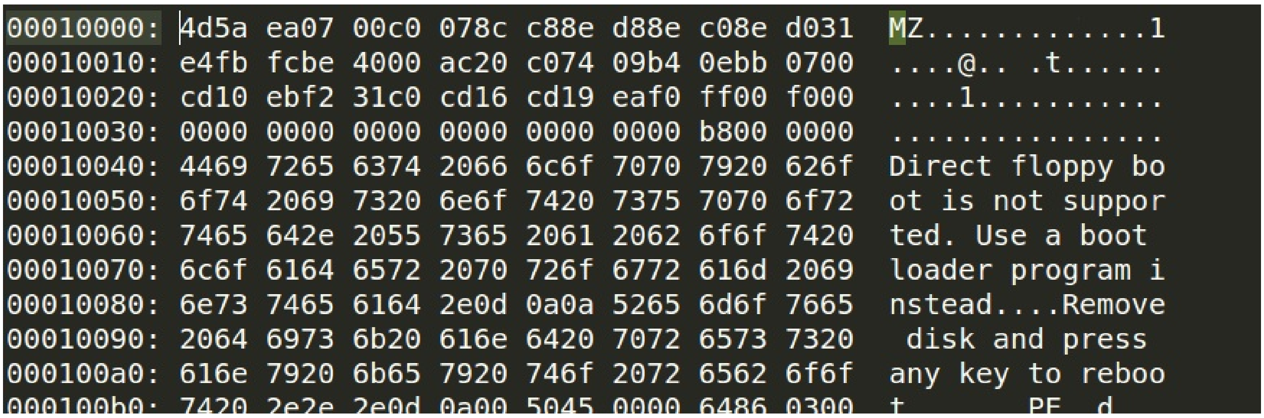
通过阅读 kernel boot protocol，在内核被引导入内存后，内存使用情况将入下表所示：

```
         | Protected-mode kernel  |
100000   +------------------------+
         | I/O memory hole        |
0A0000   +------------------------+
         | Reserved for BIOS      | Leave as much as possible unused
         ~                        ~
         | Command line           | (Can also be below the X+10000 mark)
X+10000  +------------------------+
         | Stack/heap             | For use by the kernel real-mode code.
X+08000  +------------------------+
         | Kernel setup           | The kernel real-mode code.
         | Kernel boot sector     | The kernel legacy boot sector.
       X +------------------------+
         | Boot loader            | <- Boot sector entry point 0x7C00
001000   +------------------------+
         | Reserved for MBR/BIOS  |
000800   +------------------------+
         | Typically used by MBR  |
000600   +------------------------+
         | BIOS use only          |
000000   +------------------------+
```

所以当 bootloader 完成任务，将执行权移交给 kernel，kernel 的代码从以下地址开始执行：

```
0x1000 + X + sizeof(KernelBootSector) + 1
个人以为应该是 X + sizeof(KernelBootSector) + 1 因为 X 已经是一个具体的物理地址了，不是一个偏移
```

上面的公式中， `X` 是 kernel bootsector 被引导入内存的位置。在我的机器上， `X` 的值是 `0x10000` ，我们可以通过 memory dump 来检查这个地址：



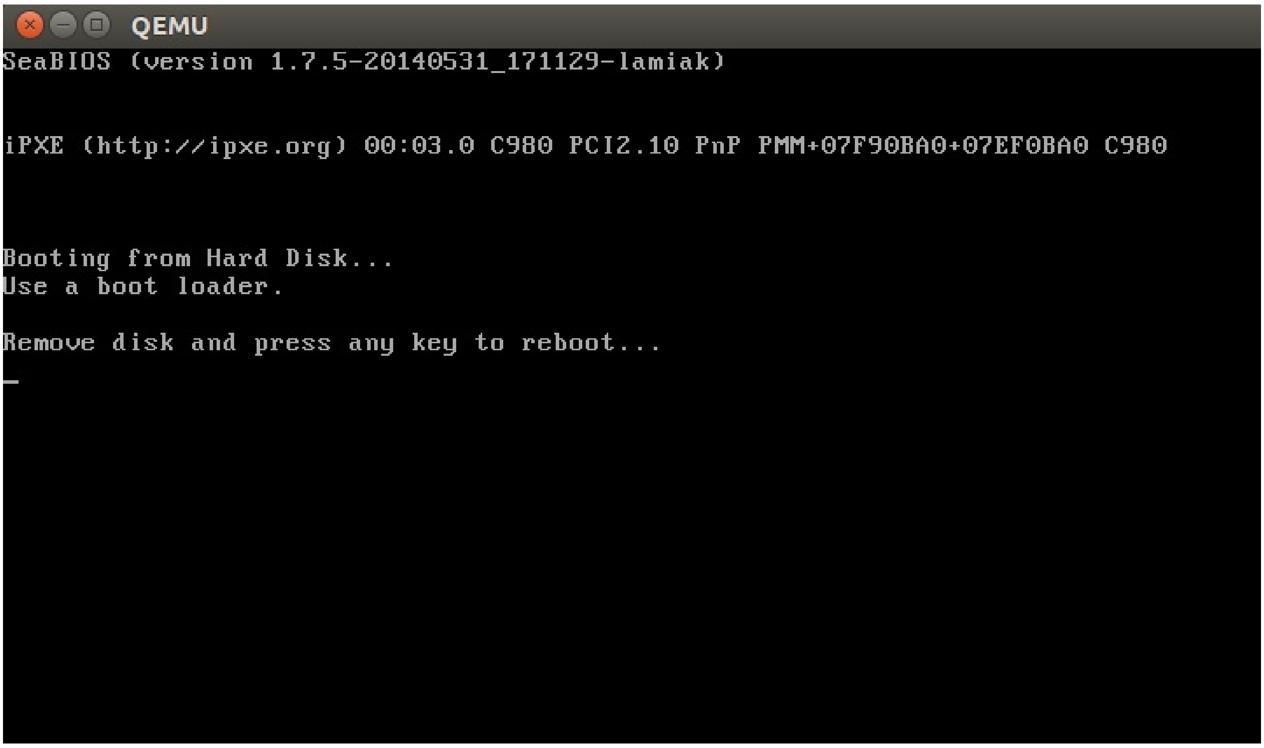
到这里，引导程序完成它的使命，并将控制权移交给了 Linux kernel。下面我们就来看看 kernel setup code 都做了些什么。

内核设置

经过上面的一系列操作，我们终于进入到内核了。不过从技术上说，内核还没有被运行起来，因为首先我们需要正确设置内核，启动内存管理，进程管理等等。内核设置代码的运行起点是 arch/x86/boot/header.S 中定义的 _start 函数。 在 `_start` 函数开始之前，还有很多的代码，那这些代码是做什么的呢？

实际上 `_start` 开始之前的代码是 kenerl 自带的 bootloader。在很久以前，是可以使用这个 bootloader 来启动 Linux 的。不过在新的 Linux 中，这个 bootloader 代码已经不再启动 Linux 内核，而只是输出一个错误信息。 如果你运行下面的命令，直接使用 Linux 内核来启动，你会看到下图所示的错误：

```
qemu-system-x86_64 vmlinuz-3.18-generic
```



为了能够作为 bootloader 来使用, `header.S` 开始处定义了 [MZ] MZ 魔术数字, 并且定义了 PE 头，在 PE 头中定义了输出的字符串：

```
#ifdef CONFIG_EFI_STUB
# "MZ", MS-DOS header
.byte 0x4d
.byte 0x5a
#endif
...
...
...
pe_header:
    .ascii "PE"
    .word 0
```

之所以代码需要这样写，这个是因为遵从 UEFI 的硬件需要这样的结构才能正常引导操作系统。

去除这些作为 bootloader 使用的代码，真正的内核代码就从 `_start` 开始了：

```
// header.S line 292
.globl _start
_start:
```

其他的 bootloader (grub2 and others) 知道 _start 所在的位置（ 从 `MZ` 头开始偏移 `0x200` 字节 ），所以这些 bootloader 就会忽略所有在这个位置前的代码（这些之前的代码位于 `.bstext` 段中）， 直接跳转到这个位置启动内核。

```
//
// arch/x86/boot/setup.ld
//
. = 0;                    // current position
.bstext : { *(.bstext) }  // put .bstext section to position 0
.bsdata : { *(.bsdata) }
```

```
    .globl _start
_start:
    .byte 0xeb
    .byte start_of_setup-1f
1:
    //
    // rest of the header
    //
```

`_start` 开始就是一个 `jmp` 语句（ `jmp` 语句的 opcode 是 `0xeb` ），这个跳转语句是一个短跳转，跟在后面的是一个相对地址（ `start_of_setup - 1f` ）。在汇编代码中 `Nf` 代表了当前代码之后第一个标号为 `N` 的代码段的地址。回到我们的代码，在 `_start` 标号之后的第一个标号为 `1` 的代码段中包含了剩下的 setup header 结构。在标号为 `1` 的代码段结束之后，紧接着就是标号为 `start_of_setup` 的代码段（这个代码段位于 `.entrytext` 代码区，这个代码段中的第一条指令实际上是内核开始执行之后的第一条指令）。

下面让我们来看一下 GRUB2 的代码是如何跳转到 `_start` 标号处的。从 Linux 内核代码中，我们知道 `_start` 标号的代码位于偏移 `0x200` 处。在 GRUB2 的源代码中我们可以看到下面的代码：

```
  state.gs = state.fs = state.es = state.ds = state.ss = segment;
  state.cs = segment + 0x20;
```

在我的机器上，因为我的内核代码被加载到了内存地址 `0x10000` 处，所以在上面的代码执行完成之后 `cs = 0x1020` （因此第一条指令的内存地址将是 `cs << 4 + 0 = 0x10200` ，刚好是 `0x10000` 开始后的 `0x200` 处的指令）：

```
fs = es = ds = ss = 0x1000
cs = 0x1020
```

从 `start_of_setup` 标号开始的代码需要完成下面这些事情：

- 将所有段寄存器的值设置成一样的内容
- 设置堆栈
- 设置 bss （静态变量区）
- 跳转到 main.c 开始执行代码

段寄存器设置

在代码的一开始，就将 `ds` 和 `es` 段寄存器的内容设置成一样，并且使用指令 `sti` 来允许中断发生：

```
    movw    %ds, %ax
    movw    %ax, %es
    sti
```

就像我在上面一节中所写的， 为了能够跳转到 `_start` 标号出执行代码，grub2 将 `cs` 段寄存器的值设置成了 `0x1020`，这个值和其他段寄存器都是不一样的，因此下面的代码就是将 `cs` 段寄存器的值和其他段寄存器一致：

```
    pushw    %ds
    pushw    $6f
    lretw
```

上面的代码使用了一个小小的技巧来重置 `cs` 寄存器的内容，下面我们就来仔细分析。 这段代码首先将 `ds` 寄存器的值入栈，然后将标号为 6 的代码段地址入栈 ，接着执行 `lretw` 指令，这条指令，将把标号为 `6` 的内存地址放入 `ip` 寄存器 （instruction pointer），将 `ds` 寄存器的值放入 `cs` 寄存器。 这样一来 `ds` 和 `cs` 段寄存器就拥有了相同的值。

设置堆栈

绝大部分的 setup 代码都是为 C 语言运行环境做准备。在设置了 `ds` 和 `es` 寄存器之后，接下来 step 的代码将检查 `ss` 寄存器的内容，如果寄存器的内容不对，那么将进行更正：

```
    movw    %ss, %dx
    cmpw    %ax, %dx
    movw    %sp, %dx
    je    2f
```

当进入这段代码的时候， `ss` 寄存器的值可能是一下三种情况之一：

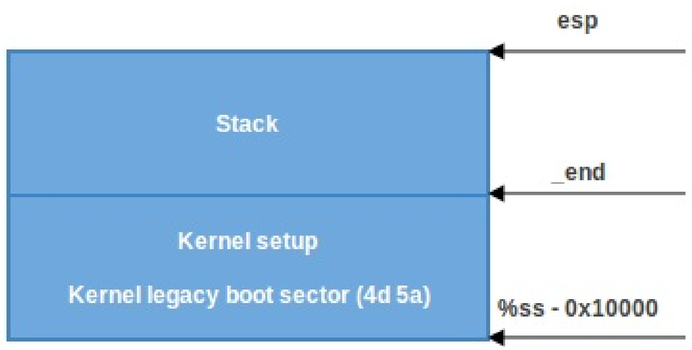
- `ss` 寄存器的值是 0x10000 (和其他除了 `cs` 寄存器之外的所有寄存器的一样）
- `ss` 寄存器的值不是 0x10000，但是 `CAN_USE_HEAP` 标志被设置了
- `ss` 寄存器的值不是 0x10000，同时 `CAN_USE_HEAP` 标志没有被设置

下面我们就来分析在这三中情况下，代码都是如何工作的：

- `ss` 寄存器的值是 0x10000，在这种情况下，代码将直接跳转到标号为 `2` 的代码处执行:

```
2:      andw    $~3, %dx
    jnz     3f
    movw    $0xfffc, %dx
3:  movw    %ax, %ss
    movzwl %dx, %esp
    sti
```

这段代码首先将 `dx` 寄存器的值（就是当前 `sp` 寄存器的值）4字节对齐，然后检查是否为0（如果是0，堆栈就不对了，因为堆栈是从大地址向小地址发展的），如果是0，那么就将 `dx` 寄存器的值设置成 `0xfffc` （64KB地址段的最后一个4字节地址）。如果不是0，那么就保持当前值不变。接下来，就将 `ax` 寄存器的值（ 0x10000 ）设置到 `ss` 寄存器，并根据 `dx` 寄存器的值设置正确的 `sp` 。这样我们就得到了正确的堆栈设置，具体请参考下图：



- 下面让我们来看 `ss` != `ds` 的情况，首先将 setup code 的结束地址 _end 写入 `dx` 寄存器。然后检查 `loadflags` 中是否设置了 `CAN_USE_HEAP` 标志。 根据 kernel boot protocol 的定义，loadflags 是一个标志字段。这个字段的 `Bit 7` 就是 `CAN_USE_HEAP` 标志：

```
Field name:     loadflags

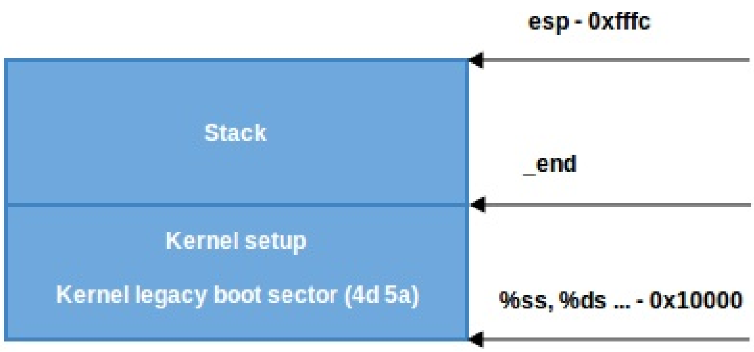
  This field is a bitmask.

  Bit 7 (write): CAN_USE_HEAP
    Set this bit to 1 to indicate that the value entered in the
    heap_end_ptr is valid.  If this field is clear, some setup code
    functionality will be disabled.
```

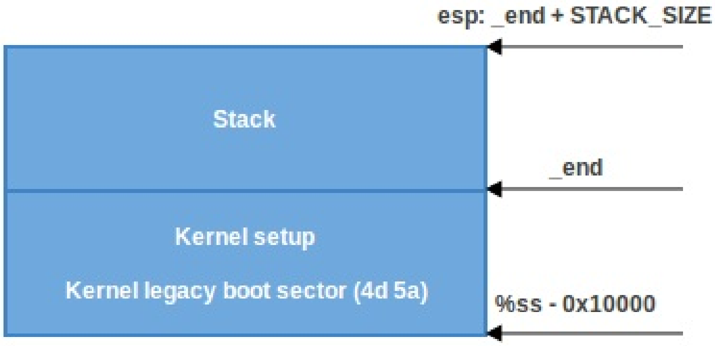
`loadflags` 字段其他可以设置的标志包括：

```
#define LOADED_HIGH         (1<<0)
#define QUIET_FLAG          (1<<5)
#define KEEP_SEGMENTS       (1<<6)
#define CAN_USE_HEAP        (1<<7)
```

如果 `CAN_USE_HEAP` 被置位，那么将 `heap_end_ptr` 放入 `dx` 寄存器，然后加上 `STACK_SIZE`（最小堆栈大小是 512 bytes）。在加法完成之后，如果结果没有溢出（CF flag 没有置位，如果置位那么程序就出错了），那么就跳转到标号为 `2` 的代码处继续执行（这段代码的逻辑在1中已经详细介绍了），接着我们就得到了如下图所示的堆栈：



- 最后一种情况就是 `CAN_USE_HEAP` 没有置位， 那么我们就将 `dx` 寄存器的值加上 `STACK_SIZE`，然后跳转到标号为 `2` 的代码处继续执行，接着我们就得到了如下图所示的堆栈：



BSS段设置

在我们正式执行 C 代码之前，我们还有2件事情需要完成。1）设置正确的 BSS段 ；2）检查 `magic` 签名。接下来的代码，首先检查 `magic` 签名 setup_sig，如果签名不对，直接跳转到 `setup_bad` 部分执行代码：

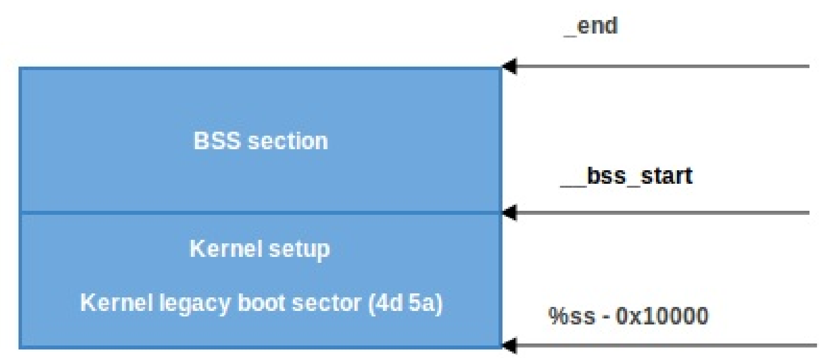
```
cmpl    $0x5a5aaa55, setup_sig
jne     setup_bad
```

如果 `magic` 签名是对的， 那么我们只要设置好 `BSS` 段，就可以开始执行 C 代码了。

BSS 段用来存储那些没有被初始化的静态变量。对于这个段使用的内存， Linux 首先使用下面的代码将其全部清零：

```
movw    $__bss_start, %di
movw    $_end+3, %cx
xorl    %eax, %eax
subw    %di, %cx
shrw    $2, %cx
rep; stosl
```

在这段代码中，首先将 __bss_start 地址放入 `di` 寄存器，然后将 `_end + 3` （4字节对齐）地址放入 `cx` ，接着使用 `xor` 指令将 `ax` 寄存器清零，接着计算 BSS 段的大小（ `cx` - `di` ），然后将大小放入 `cx` 寄存器。接下来将 `cx` 寄存器除4，最后使用 `rep; stosl` 指令将 `ax` 寄存器的值（0）写入 寄存器整个 BSS 段。 代码执行完成之后，我们将得到如下图所示的 BSS 段:



跳转到 main 函数

到目前为止，我们完成了堆栈和 BSS 的设置，现在我们可以正式跳入 `main()` 函数来执行 C 代码了：

```
calll main
```

`main()` 函数定义在 arch/x86/boot/main.c，我们将在下一章详细介绍这个函数做了什么事情。

结束语

本章到此结束了，在下一章中我们将详细介绍在 Linux 内核设置过程中调用的第一个 C 代码（ `main()` ），也将介绍诸如 `memset`, `memcpy`, `earlyprintk` 这些底层函数的实现，敬请期待。

如果你有任何的问题或者建议，你可以留言，也可以直接发消息给我twitter。

如果你发现文中描述有任何问题，请提交一个 **PR** 到 **linux-insides-zh** 。

相关链接

- Intel 80386 programmer's reference manual 1986
- Minimal Boot Loader for Intel® Architecture
- 8086
- 80386
- Reset vector
- Real mode
- Linux kernel boot protocol
- CoreBoot developer manual
- Ralf Brown's Interrupt List
- Power supply
- Power good signal

在内核安装代码的第一步

内核启动的第一步

在上一节中我们开始接触到内核启动代码，并且分析了初始化部分，最后我们停在了对 `main` 函数（ `main` 函数是第一个用C写的函数）的调用（ `main` 函数位于 arch/x86/boot/main.c）。

在这一节中我们将继续对内核启动过程的研究，我们将

- 认识 `保护模式`
- 如何从实模式进入保护模式
- 堆和控制台初始化
- 内存检测，cpu验证，键盘初始化
- 还有更多

现在让我们开始我们的旅程

保护模式

在操作系统可以使用Intel 64位CPU的长模式之前，内核必须首先将CPU切换到保护模式运行。

什么是保护模式？保护模式于1982年被引入到Intel CPU家族，并且从那之后，直到Intel 64出现，保护模式都是Intel CPU的主要运行模式。

淘汰实模式的主要原因是因为在实模式下，系统能够访问的内存非常有限。如果你还记得我们在上一节说的，在实模式下，系统最多只能访问1M内存，而且在很多时候，实际能够访问的内存只有640K。

保护模式带来了很多的改变，不过主要的改变都集中在内存管理方法。在保护模式中，实模式的20位地址线被替换成32位地址线，因此系统可以访问多达4GB的地址空间。另外，在保护模式中引入了内存分页功能，在后面的章节中我们将介绍这个功能。

保护模式提供了2种完全不同的内存管理机制：

- 段式内存管理
- 内存分页

在这一节中，我们只介绍段式内存管理，内存分页我们将在后面的章节进行介绍。

在上一节中我们说过，在实模式下，一个物理地址是由2个部分组成的：

- 内存段的基地址
- 从基地址开始的偏移

使用这2个信息，我们可以通过下面的公式计算出对应的物理地址

```
PhysicalAddress = Segment * 16 + Offset
```

在保护模式中，内存段的定义和实模式完全不同。在保护模式中，每个内存段不再是64K大小，段的大小和起始位置是通过一个叫做`段描述符`的数据结构进行描述。所有内存段的段描述符存储在一个叫做`全局描述符表`(GDT)的内存结构中。

`全局描述符表`这个内存数据结构在内存中的位置并不是固定的，它的地址保存在一个特殊寄存器`GDTR`中。在后面的章节中，我们将在Linux内核代码中看到全局描述符表的地址是如何被保存到`GDTR`中的。具体的汇编代码看起来是这样的：

```
lgdt gdt
```

`lgdt`汇编代码将把全局描述符表的基地址和大小保存到`GDTR`寄存器中。`GDTR`是一个48位的寄存器，这个寄存器中的保存了2部分的内容:

- 全局描述符表的大小 (16位）
- 全局描述符表的基址 (32位)

就像前面的段落说的，全局描述符表包含了所有内存段的`段描述符`。每个段描述符长度是64位，结构如下图描述：

```
31          24        19      16              7            0
------------------------------------------------------------
|             | |B| |A|       | |   | |0|E|W|A|            |
| BASE 31:24  |G|/|L|V| LIMIT |P|DPL|S|  TYPE | BASE 23:16 | 4
|             | |D| |L| 19:16 | |   | |1|C|R|A|            |
------------------------------------------------------------
|                             |                            |
|        BASE 15:0            |       LIMIT 15:0           | 0
|                             |                            |
------------------------------------------------------------
```

粗粗一看，上面的结构非常吓人，不过实际上这个结构是非常容易理解的。比如在上图中的LIMIT 15:0 表示这个数据结构的0到15位保存的是内存段的大小的0到15位。相似的 LIMITE 19:16 表示上述数据结构的16到19位保存的是内存段大小的16到19位。从这个分析中，我们可以看出每个内存段的大小是通过20位进行描述的。下面我们将对这个数据结构进行仔细分析：

1. Limit[20位] 被保存在上述内存结构的0-15和16-19位。根据上述内存结构中 `G` 位的设置，这20位内存定义的内存长度是不一样的。下面是一些具体的例子：

    - 如果 `G` = 0, 并且Limit = 0， 那么表示段长度是1 byte
    - 如果 `G` = 1, 并且Limit = 0, 那么表示段长度是4K bytes
    - 如果 `G` = 0，并且Limit = 0xfffff，那么表示段长度是1M bytes
    - 如果 `G` = 1，并且Limit = 0xfffff，那么表示段长度是4G bytes

    从上面的例子我们可以看出：

    - 如果G = 0, 那么内存段的长度是按照1 byte进行增长的 (Limit每增加1，段长度增加1 byte)，最大的内存段长度将是1M bytes；
    - 如果G = 1, 那么内存段的长度是按照4K bytes进行增长的 (Limit每增加1，段长度增加4K bytes)，最大的内存段长度将是4G bytes;
    - 段长度的计算公式是 base_seg_length * (LIMIT + 1)。
2. Base[32-bits] 被保存在上述地址结构的0-15， 32-39以及56-63位。Base定义了段基址。

3. Type/Attribute (40-47 bits) 定义了内存段的类型以及支持的操作。

    - `S` 标记（ 第44位 ）定义了段的类型，`S` = 0说明这个内存段是一个系统段；`S` = 1说明这个内存段是一个代码段或者是数据段（ 堆栈段是一种特殊类型的数据段，堆栈段必须是可以进行读写的段 ）。

在 `S` = 1的情况下，上述内存结构的第43位决定了内存段是数据段还是代码段。如果43位 = 0，说明是一个数据段，否则就是一个代码段。

对于数据段和代码段，下面的表格给出了段类型定义

```
|          Type Field          | Descriptor Type | Description
|-----------------------------|-----------------|-------------------
| Decimal                     |                 |
|             0    E    W   A |                 |
| 0           0    0    0   0 | Data            | Read-Only
| 1           0    0    0   1 | Data            | Read-Only, accessed
| 2           0    0    1   0 | Data            | Read/Write
| 3           0    0    1   1 | Data            | Read/Write, accessed
| 4           0    1    0   0 | Data            | Read-Only, expand-down
| 5           0    1    0   1 | Data            | Read-Only, expand-down, accessed
| 6           0    1    1   0 | Data            | Read/Write, expand-down
| 7           0    1    1   1 | Data            | Read/Write, expand-down, accessed
|                  C    R   A |                 |
| 8           1    0    0   0 | Code            | Execute-Only
| 9           1    0    0   1 | Code            | Execute-Only, accessed
| 10          1    0    1   0 | Code            | Execute/Read
| 11          1    0    1   1 | Code            | Execute/Read, accessed
| 12          1    1    0   0 | Code            | Execute-Only, conforming
| 14          1    1    0   1 | Code            | Execute-Only, conforming, accessed
| 13          1    1    1   0 | Code            | Execute/Read, conforming
| 15          1    1    1   1 | Code            | Execute/Read, conforming, accessed
```

从上面的表格我们可以看出，当第43位是 `0` 的时候，这个段描述符对应的是一个数据段，如果该位是 `1` ，那么表示这个段描述符对应的是一个代码段。对于数据段，第42，41，40位表示的是(*E*扩展，*W*可写，*A*可访问）；对于代码段，第42，41，40位表示的是(*C*一致，*R*可读，*A*可访问）。

- 如果 `E` = 0，数据段是向上扩展数据段，反之为向下扩展数据段。关于向上扩展和向下扩展数据段，可以参考下面的链接。在一般情况下，应该是不会使用向下扩展数据段的。
- 如果 `W` = 1，说明这个数据段是可写的，否则不可写。所有数据段都是可读的。
- A位表示该内存段是否已经被CPU访问。
- 如果 `C` = 1，说明这个代码段可以被低优先级的代码访问，比如可以被用户态代码访问。反之如果 `C` = 0，说明只能同优先级的代码段可以访问。
- 如果 `R` = 1，说明该代码段可读。代码段是永远没有写权限的。

1. DPL（2-bits, bit 45 和 46）定义了该段的优先级。具体数值是0-3。
2. P 标志(bit 47) - 说明该内存段是否已经存在于内存中。如果 `P` = 0，那么在访问这个内存段的时候将报错。
3. AVL 标志(bit 52) - 这个位在Linux内核中没有被使用。
4. L 标志(bit 53) - 只对代码段有意义，如果 `L` = 1，说明该代码段需要运行在64位模式下。

5. D/B flag(bit 54) - 根据段描述符描述的是一个可执行代码段、下扩数据段还是一个堆栈段，这个标志具有不同的功能。（对于32位代码和数据段，这个标志应该总是设置为1；对于16位代码和数据段，这个标志被设置为0。）。
    - 可执行代码段。此时这个标志称为D标志并用于指出该段中的指令引用有效地址和操作数的默认长度。如果该标志置位，则默认值是32位地址和32位或8位的操作数；如果该标志为0，则默认值是16位地址和16位或8位的操作数。指令前缀0x66可以用来选择非默认值的操作数大小；前缀0x67可用来选择非默认值的地址大小。
    - 栈段（由SS寄存器指向的数据段）。此时该标志称为B（Big）标志，用于指明隐含堆栈操作（如PUSH、POP或CALL）时的栈指针大小。如果该标志置位，则使用32位栈指针并存放在ESP寄存器中；如果该标志为0，则使用16位栈指针并存放在SP寄存器中。如果堆栈段被设置成一个下扩数据段，这个B标志也同时指定了堆栈段的上界限。
    - 下扩数据段。此时该标志称为B标志，用于指明堆栈段的上界限。如果设置了该标志，则堆栈段的上界限是0xFFFFFFFF（4GB）；如果没有设置该标志，则堆栈段的上界限是0xFFFF（64KB）。

在保护模式下，段寄存器保存的不再是一个内存段的基地址，而是一个称为 `段选择子` 的结构。每个段描述符都对应一个 `段选择子` 。 `段选择子` 是一个16位的数据结构，下图显示了这个数据结构的内容：

```
-----------------------------
|       Index    | TI | RPL |
-----------------------------
```

其中，

- **Index** 表示在GDT中，对应段描述符的索引号。
- **TI** 表示要在GDT还是LDT中查找对应的段描述符
- **RPL** 表示请求者优先级。这个优先级将和段描述符中的优先级协同工作，共同确定访问是否合法。

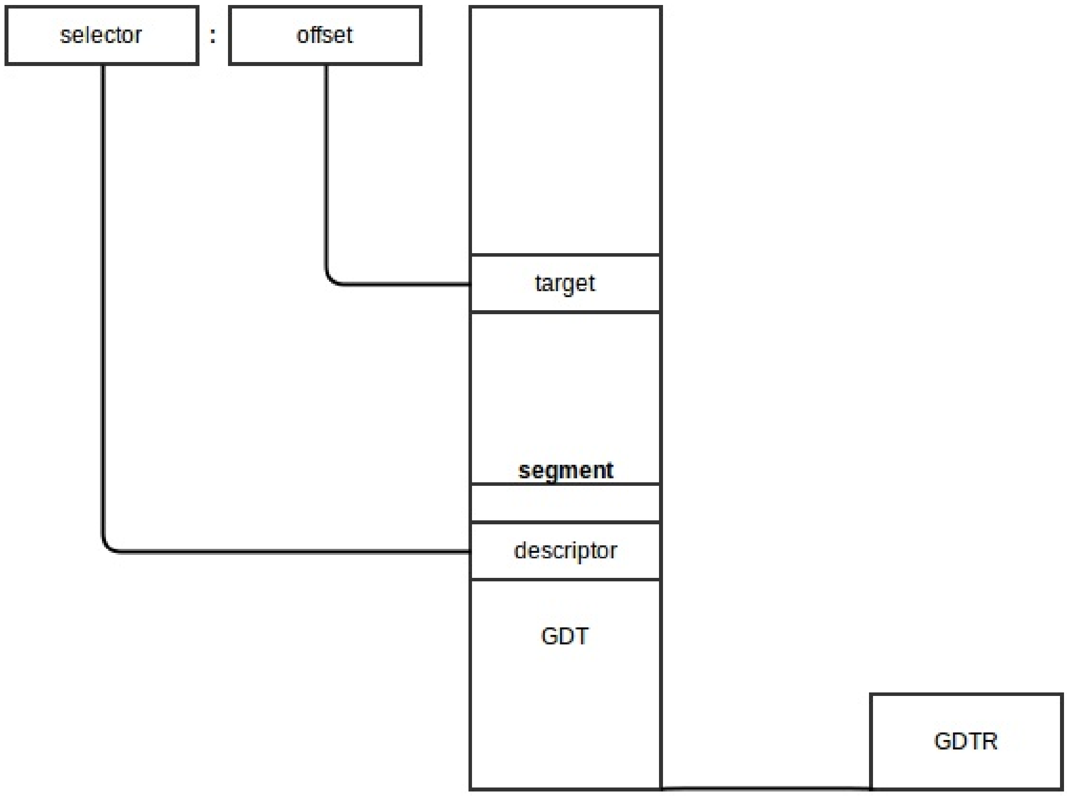
在保护模式下，每个段寄存器实际上包含下面2部分内容：

- 可见部分 - 段选择子
- 隐藏部分 - 段描述符

在保护模式中，cpu是通过下面的步骤来找到一个具体的物理地址的：

- 代码必须将相应的 `段选择子` 装入某个段寄存器
- CPU根据 `段选择子` 从GDT中找到一个匹配的段描述符，然后将段描述符放入段寄存器的隐藏部分
- 在没有使用向下扩展段的时候，那么内存段的基地址就是 `段描述符中的基地址` ，段描述符的 `limit + 1` 就是内存段的长度。如果你知道一个内存地址的 `偏移` ，那么在没有开启分

页机制的情况下，这个内存的物理地址就是`基地址+偏移`



当代码要从实模式进入保护模式的时候，需要执行下面的操作：

- 禁止中断发生
- 使用命令 `lgdt` 将GDT表装入 `GDTR` 寄存器
- 设置CR0寄存器的PE位为1，使CPU进入保护模式
- 跳转开始执行保护模式代码

在后面的章节中，我们将看到Linux 内核中完整的转换代码。不过在系统进入保护模式之前，内核有很多的准备工作需要进行。

让我们打开C文件 arch/x86/boot/main.c。这个文件包含了很多的函数，这些函数分别会执行键盘初始化，内存堆初始化等等操作...，下面让我们来具体看一些重要的函数。

将启动参数拷贝到"zeropage"

让我们从 `main` 函数开始看起，这个函数中，首先调用了 `copy_boot_params(void)` 。

这个函数将内核设置信息拷贝到 `boot_params` 结构的相应字段。大家可以在arch/x86/include/uapi/asm/bootparam.h找到 `boot_params` 结构的定义。

`boot_params` 结构中包含 `struct setup_header hdr` 字段。这个结构包含了linux boot protocol中定义的相同字段，并且由boot loader填写。在内核编译的时候 `copy_boot_params` 完成两个工作：

1. 将header.S中定义的 `hdr` 结构中的内容拷贝到 `boot_params` 结构的字段 `struct setup_header hdr` 中。
2. 如果内核是通过老的命令行协议运行起来的，那么就更新内核的命令行指针。

这里需要注意的是拷贝 `hdr` 数据结构的 `memcpy` 函数不是C语言中的函数，而是定义在copy.S。让我们来具体分析一下这段代码：

```
GLOBAL(memcpy)
    pushw   %si          ;push si to stack
    pushw   %di          ;push di to stack
    movw    %ax, %di     ;move &boot_param.hdr to di
    movw    %dx, %si     ;move &hdr to si
    pushw   %cx          ;push cx to stack ( sizeof(hdr) )
    shrw    $2, %cx
    rep; movsl           ;copy based on 4 bytes
    popw    %cx          ;pop cx
    andw    $3, %cx      ;cx = cx % 4
    rep; movsb           ;copy based on one byte
    popw    %di
    popw    %si
    retl
ENDPROC(memcpy)
```

在 `copy.S` 文件中，你可以看到所有的方法都开始于 `GLOBAL` 宏定义，而结束于 `ENDPROC` 宏定义。

你可以在 arch/x86/include/asm/linkage.h中找到 `GLOBAL` 宏定义。这个宏给代码段分配了一个名字标签，并且让这个名字全局可用。

```
#define GLOBAL(name)    \
    .globl name;    \
    name:
```

你可以在include/linux/linkage.h中找到 `ENDPROC` 宏的定义。 这个宏通过 `END(name)` 代码标识了汇编函数的结束，同时将函数名输出，从而静态分析工具可以找到这个函数。

```
#define ENDPROC(name) \
    .type name, @function ASM_NL \
    END(name)
```

`memcpy` 的实现代码是很容易理解的。首先，代码将 `si` 和 `di` 寄存器的值压入堆栈进行保存，这么做的原因是因为后续的代码将修改 `si` 和 `di` 寄存器的值。`memcpy` 函数（也包括其他定义在copy.s中的其他函数）使用了 `fastcall` 调用规则，意味着所有的函数调用参数是通过 `ax`, `dx`, `cx` 寄存器传入的，而不是传统的通过堆栈传入。因此在使用下面的代码调用 `memcpy` 函数的时候

```
memcpy(&boot_params.hdr, &hdr, sizeof hdr);
```

函数的参数是这样传递的

- `ax` 寄存器指向 `boot_param.hdr` 的内存地址
- `dx` 寄存器指向 `hdr` 的内存地址
- `cx` 寄存器包含 `hdr` 结构的大小

`memcpy` 函数在将 `si` 和 `di` 寄存器压栈之后，将 `boot_param.hdr` 的地址放入 `di` 寄存器，将 `hdr` 的地址放入 `si` 寄存器，并且将 `hdr` 数据结构的大小压栈。 接下来代码首先以4个字节为单位，将 `si` 寄存器指向的内存内容拷贝到 `di` 寄存器指向的内存。当剩下的字节数不足4字节的时候，代码将原始的 `hdr` 数据结构大小出栈放入 `cx` ，然后对 `cx` 的值对4求模，接下来就是根据 `cx` 的值，以字节为单位将 `si` 寄存器指向的内存内容拷贝到 `di` 寄存器指向的内存。当拷贝操作完成之后，将保留的 `si` 以及 `di` 寄存器值出栈，函数返回。

控制台初始化

在 `hdr` 结构体被拷贝到 `boot_params.hdr` 成员之后，系统接下来将进行控制台的初始化。控制台初始化时通过调用arch/x86/boot/early_serial_console.c中定义的 `console_init` 函数实现的。

这个函数首先查看命令行参数是否包含 `earlyprintk` 选项。如果命令行参数包含该选项，那么函数将分析这个选项的内容。得到控制台将使用的串口信息，然后进行串口的初始化。以下是 `earlyprintk` 选项可能的取值：

- serial,0x3f8,115200
- serial,ttyS0,115200
- ttyS0,115200

当串口初始化成功之后，如果命令行参数包含 `debug` 选项，我们将看到如下的输出。

```
if (cmdline_find_option_bool("debug"))
    puts("early console in setup code\n");
```

`puts` 函数定义在tty.c。这个函数只是简单的调用 `putchar` 函数将输入字符串中的内容按字节输出。下面让我们来看看 `putchar` 函数的实现：

```
void __attribute__((section(".inittext"))) putchar(int ch)
{
    if (ch == '\n')
        putchar('\r');

    bios_putchar(ch);

    if (early_serial_base != 0)
        serial_putchar(ch);
}
```

`__attribute__((section(".inittext")))` 说明这段代码将被放入 `.inittext` 代码段。关于 `.inittext` 代码段的定义你可以在 setup.ld 中找到。

如果需要输出的字符是 `\n` ，那么 `putchar` 函数将调用自己首先输出一个字符 `\r` 。接下来，就调用 `bios_putchar` 函数将字符输出到显示器（使用bios int10中断）：

```
static void __attribute__((section(".inittext"))) bios_putchar(int ch)
{
    struct biosregs ireg;

    initregs(&ireg);
    ireg.bx = 0x0007;
    ireg.cx = 0x0001;
    ireg.ah = 0x0e;
    ireg.al = ch;
    intcall(0x10, &ireg, NULL);
}
```

在上面的代码中 `initreg` 函数接受一个 `biosregs` 结构的地址作为输入参数，该函数首先调用 `memset` 函数将 `biosregs` 结构体所有成员清0。

```
    memset(reg, 0, sizeof *reg);
    reg->eflags |= X86_EFLAGS_CF;
    reg->ds = ds();
    reg->es = ds();
    reg->fs = fs();
    reg->gs = gs();
```

下面让我们来看看memset函数的实现：

```
GLOBAL(memset)
    pushw   %di
    movw    %ax, %di
    movzbl  %dl, %eax
    imull   $0x01010101,%eax
    pushw   %cx
    shrw    $2, %cx
    rep; stosl
    popw    %cx
    andw    $3, %cx
    rep; stosb
    popw    %di
    retl
ENDPROC(memset)
```

首先你会发现， `memset` 函数和 `memcpy` 函数一样使用了 `fastcall` 调用规则，因此函数的参数是通过 `ax` ， `dx` 以及 `cx` 寄存器传入函数内部的。

就像memcpy函数一样， `memset` 函数一开始将 `di` 寄存器入栈，然后将 `biosregs` 结构的地址从 `ax` 寄存器拷贝到 `di` 寄存器。接下来，使用 `movzbl` 指令将 `dl` 寄存器的内容拷贝到 `ax` 寄存器的低字节，到这里 `ax` 寄存器就包含了需要拷贝到 `di` 寄存器所指向的内存的值。

接下来的 `imull` 指令将 `eax` 寄存器的值乘上 `0x01010101` 。这么做的原因是代码每次将尝试拷贝4个字节内存的内容。下面让我们来看一个具体的例子，假设我们需要将 `0x7` 这个数值放到内存中，在执行 `imull` 指令之前， `eax` 寄存器的值是 `0x7` ，在 `imull` 指令被执行之后， `eax` 寄存器的内容变成了 `0x07070707` （4个字节的 `0x7` ）。在 `imull` 指令之后，代码使用 `rep; stosl` 指令将 `eax` 寄存器的内容拷贝到 `es:di` 指向的内存。

在 `bisoregs` 结构体被 `initregs` 函数正确填充之后， `bios_putchar` 调用中断 0x10 在显示器上输出一个字符。接下来 `putchar` 函数检查是否初始化了串口，如果串口被初始化了，那么将调用serial_putchar将字符输出到串口。

堆初始化

当堆栈和bss段在header.S中被初始化之后 (细节请参考上一篇part)， 内核需要初始化全局堆，全局堆的初始化是通过 `init_heap` 函数实现的。

代码首先检查内核设置头中的 `loadflags` 是否设置了 `CAN_USE_HEAP` 标志。 如果该标记被设置了，那么代码将计算堆栈的结束地址：:

```
    char *stack_end;

    //%P1 is (-STACK_SIZE)
    if (boot_params.hdr.loadflags & CAN_USE_HEAP) {
        asm("leal %P1(%%esp),%0"
            : "=r" (stack_end) : "i" (-STACK_SIZE));
```

换言之 `stack_end = esp - STACK_SIZE`.

在计算了堆栈结束地址之后，代码计算了堆的结束地址：

```

    //heap_end = heap_end_ptr + 512
    heap_end = (char *)((size_t)boot_params.hdr.heap_end_ptr + 0x200);
```

接下来代码判断 `heap_end` 是否大于 `stack_end`，如果条件成立，将 `stack_end` 设置成 `heap_end`（这么做是因为在大部分系统中全局堆和堆栈是相邻的，但是增长方向是相反的）。

到这里为止，全局堆就被正确初始化了。在全局堆被初始化之后，我们就可以使用 `GET_HEAP` 方法。至于这个函数的实现和使用，我们将在后续的章节中看到。

检查CPU类型

在堆栈初始化之后，内核代码通过调用arch/x86/boot/cpu.c提供的 `validate_cpu` 方法检查CPU级别以确定系统是否能够在当前的CPU上运行。

`validate_cpu` 调用了 `check_cpu` 方法得到当前系统的CPU级别，并且和系统预设的最低CPU级别进行比较。如果不满足条件，则不允许系统运行。

```
/*from cpu.c*/
check_cpu(&cpu_level, &req_level, &err_flags);
/*after check_cpu call, req_level = req_level defined in cpucheck.c*/
if (cpu_level < req_level) {
    printf("This kernel requires an %s CPU, ", cpu_name(req_level));
    printf("but only detected an %s CPU.\n", cpu_name(cpu_level));
    return -1;
}
```

除此之外， `check_cpu` 方法还做了大量的其他检测和设置工作，下面就简单介绍一些：1）检查cpu标志，如果cpu是64位cpu，那么就设置long mode, 2) 检查CPU的制造商，根据制造商的不同，设置不同的CPU选项。比如对于AMD出厂的cpu，如果不支持 `SSE+SSE2` ，那么就禁止这些选项。

内存分布侦测

接下来，内核调用 `detect_memory` 方法进行内存侦测，以得到系统当前内存的使用分布。该方法使用多种编程接口，包括 `0xe820`（获取全部内存分配），`0xe801` 和 `0x88`（获取临近内存大小），进行内存分布侦测。在这里我们只介绍arch/x86/boot/memory.c中提供的 `detect_memory_e820` 方法。

该方法首先调用 `initregs` 方法初始化 `biosregs` 数据结构，然后向该数据结构填入 `0xe820` 编程接口所要求的参数：

```
initregs(&ireg);
ireg.ax  = 0xe820;
ireg.cx  = sizeof buf;
ireg.edx = SMAP;
ireg.di  = (size_t)&buf;
```

- `ax` 固定为 `0xe820`
- `cx` 包含数据缓冲区的大小，该缓冲区将包含系统内存的信息数据
- `edx` 必须是 `SMAP` 这个魔术数字，就是 `0x534d4150`
- `es:di` 包含数据缓冲区的地址
- `ebx` 必须为0.

接下来就是通过一个循环来收集内存信息了。每个循环都开始于一个 `0x15` 中断调用，这个中断调用返回地址分配表中的一项，接着程序将返回的 `ebx` 设置到 `biosregs` 数据结构中，然后进行下一次的 `0x15` 中断调用。那么循环什么时候结束呢？直到 `0x15` 调用返回的eflags包含标志 `X86_EFLAGS_CF`：

```
intcall(0x15, &ireg, &oreg);
ireg.ebx = oreg.ebx;
```

在循环结束之后，整个内存分配信息将被写入到 `e820entry` 数组中，这个数组的每个元素包含下面3个信息:

- 内存段的起始地址
- 内存段的大小
- 内存段的类型（类型可以是reserved, usable等等)。

你可以在 `dmesg` 输出中看到这个数组的内容：

```
[    0.000000] e820: BIOS-provided physical RAM map:
[    0.000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009fbff] usable
[    0.000000] BIOS-e820: [mem 0x000000000009fc00-0x000000000009ffff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000000f0000-0x00000000000fffff] reserved
[    0.000000] BIOS-e820: [mem 0x0000000000100000-0x000000003ffdffff] usable
[    0.000000] BIOS-e820: [mem 0x000000003ffe0000-0x000000003fffffff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000fffc0000-0x00000000ffffffff] reserved
```

键盘初始化

接下来内核调用 `keyboard_init()` 方法进行键盘初始化操作。 首先，方法调用 `initregs` 初始化寄存器结构，然后调用0x16中断来获取键盘状态。

```
    initregs(&ireg);
    ireg.ah = 0x02;     /* Get keyboard status */
    intcall(0x16, &ireg, &oreg);
    boot_params.kbd_status = oreg.al;
```

在获取了键盘状态之后，代码再次调用0x16中断来设置键盘的按键检测频率。

```
    ireg.ax = 0x0305;   /* Set keyboard repeat rate */
    intcall(0x16, &ireg, NULL);
```

系统参数查询

接下来内核将进行一系列的参数查询。我们在这里将不深入介绍所有这些查询，我们将在后续章节中再进行详细介绍。在这里我们将简单介绍一些系统参数查询:

query_mca 方法调用0x15中断来获取机器的型号信息，BIOS版本以及其他一些硬件相关的属性：

```
int query_mca(void)
{
    struct biosregs ireg, oreg;
    u16 len;

    initregs(&ireg);
    ireg.ah = 0xc0;
    intcall(0x15, &ireg, &oreg);

    if (oreg.eflags & X86_EFLAGS_CF)
        return -1;  /* No MCA present */

    set_fs(oreg.es);
    len = rdfs16(oreg.bx);

    if (len > sizeof(boot_params.sys_desc_table))
        len = sizeof(boot_params.sys_desc_table);

    copy_from_fs(&boot_params.sys_desc_table, oreg.bx, len);
    return 0;
}
```

这个方法设置 `ah` 寄存器的值为 `0xc0`，然后调用 `0x15` BIOS中断。中断返回之后代码检查 carry flag。如果它被置位，说明BIOS不支持**MCA**。如果CF被设置成0，那么 `ES:BX` 指向系统信息表。这个表的内容如下所示：

```
Offset  Size    Description
 00h    WORD    number of bytes following
 02h    BYTE    model (see #00515)
 03h    BYTE    submodel (see #00515)
 04h    BYTE    BIOS revision: 0 for first release, 1 for 2nd, etc.
 05h    BYTE    feature byte 1 (see #00510)
 06h    BYTE    feature byte 2 (see #00511)
 07h    BYTE    feature byte 3 (see #00512)
 08h    BYTE    feature byte 4 (see #00513)
 09h    BYTE    feature byte 5 (see #00514)
---AWARD BIOS---
 0Ah  N BYTEs   AWARD copyright notice
---Phoenix BIOS---
 0Ah    BYTE    ??? (00h)
 0Bh    BYTE    major version
 0Ch    BYTE    minor version (BCD)
 0Dh  4 BYTEs   ASCIZ string "PTL" (Phoenix Technologies Ltd)
---Quadram Quad386---
 0Ah 17 BYTEs   ASCII signature string "Quadram Quad386XT"
---Toshiba (Satellite Pro 435CDS at least)---
 0Ah  7 BYTEs   signature "TOSHIBA"
 11h    BYTE    ??? (8h)
 12h    BYTE    ??? (E7h) product ID??? (guess)
 13h  3 BYTEs   "JPN"
```

接下来代码调用 `set_fs` 方法，将 `es` 寄存器的值写入 `fs` 寄存器:

```
static inline void set_fs(u16 seg)
{
    asm volatile("movw %0,%%fs" : : "rm" (seg));
}
```

在boot.h 存在很多类似于 `set_fs` 的方法, 比如 `set_gs` 。

在 `query_mca` 的最后，代码将 `es:bx` 指向的内存地址的内容拷贝到 `boot_params.sys_desc_table` 。

接下来，内核调用 `query_ist` 方法获取Intel SpeedStep信息。这个方法首先检查CPU类型，然后调用 `0x15` 中断获得这个信息并放入 `boot_params` 中。

接下来，内核会调用query_apm_bios 方法从BIOS获得 高级电源管理 信息。`query_apm_bios` 也是调用 `0x15` 中断，只不过将 `ax` 设置成 `0x5300` 以得到APM设置信息。中断调用返回之后，代码将检查 `bx` 和 `cx` 的值，如果 `bx` 不是 `0x504d` (PM 标记)，或者 `cx` 不是 `0x02` (0x02，表示支持32位模式)，那么代码直接返回错误。否则，将进行下面的步骤。

接下来，代码使用 `ax = 0x5304` 来调用 `0x15` 中断，以断开 `APM` 接口；然后使用 `ax = 0x5303` 调用 `0x15` 中断，使用32位接口重新连接 `APM` ；最后使用 `ax = 0x5300` 调用 `0x15` 中断再次获取APM设置，然后将信息写入 `boot_params.apm_bios_info` 。

需要注意的是，只有在 `CONFIG_APM` 或者 `CONFIG_APM_MODULE` 被设置的情况下， `query_apm_bios` 方法才会被调用：

```
#if defined(CONFIG_APM) || defined(CONFIG_APM_MODULE)
    query_apm_bios();
#endif
```

最后是 `query_edd` 方法调用, 这个方法从BIOS中查询 `Enhanced Disk Drive` 信息。下面让我们看看 `query_edd` 方法的实现。

首先，代码检查内核命令行参数是否设置了edd 选项，如果edd选项设置成 `off` ， `query_edd` 不做任何操作，直接返回。

如果EDD被激活了， `query_edd` 遍历所有BIOS支持的硬盘，并获取相应硬盘的EDD信息：

```
for (devno = 0x80; devno < 0x80+EDD_MBR_SIG_MAX; devno++) {
    if (!get_edd_info(devno, &ei) && boot_params.eddbuf_entries < EDDMAXNR) {
        memcpy(edp, &ei, sizeof ei);
        edp++;
        boot_params.eddbuf_entries++;
    }
    ...
    ...
    ...
```

在代码中 `0x80` 是第一块硬盘， `EDD_MBR_SIG_MAX` 是一个宏，值为16。代码把获得的信息放入数组edd_info中。 `get_edd_info` 方法通过调用 `0x13` 中断调用（设置 `ah = 0x41`) 来检查EDD是否被硬盘支持。如果EDD被支持，代码将再次调用 `0x13` 中断，在这次调用中 `ah = 0x48` ，并且 `si` 指向一个数据缓冲区地址。中断调用之后，EDD信息将被保存到 `si` 指向的缓冲区地址。

结束语

本章到此就结束了，在下一章我们将讲解显示模式设置，以及在进入保护模式之前的其他准备工作，在下一章的最后我们将成功进入保护模式。

如果你有任何的问题或者建议，你可以留言，也可以直接发消息给我twitter.

如果你发现文中描述有任何问题，请提交一个 **PR** 到 **linux-insides-zh** 。

相关链接

- Protected mode

* Protected mode
* Long mode
* Nice explanation of CPU Modes with code
* How to Use Expand Down Segments on Intel 386 and Later CPUs
* earlyprintk documentation
* Kernel Parameters
* Serial console
* Intel SpeedStep
* APM
* EDD specification
* TLDP documentation for Linux Boot Process (old)
* Previous Part
* BIOS Interrupt

内核启动过程，第三部分

显示模式初始化和进入保护模式

这一章是 `内核启动过程` 的第三部分，在前一章中，我们的内核启动过程之旅停在了对 `set_video` 函数的调用（这个函数定义在 main.c）。在这一章中，我们将接着上一章继续我们的内核启动之旅。在这一章你将读到下面的内容：

- 显示模式的初始化，
- 在进入保护模式之前的准备工作，
- 正式进入保护模式

注意 如果你对保护模式一无所知，你可以查看前一章 的相关内容。另外，你也可以查看下面这些链接 以了解更多关于保护模式的内容。

就像我们前面所说的，我们将从 `set_video` 函数开始我们这章的内容，你可以在 arch/x86/boot/video.c 找到这个函数的定义。 这个函数首先从 `boot_params.hdr` 数据结构获取显示模式设置：

```
u16 mode = boot_params.hdr.vid_mode;
```

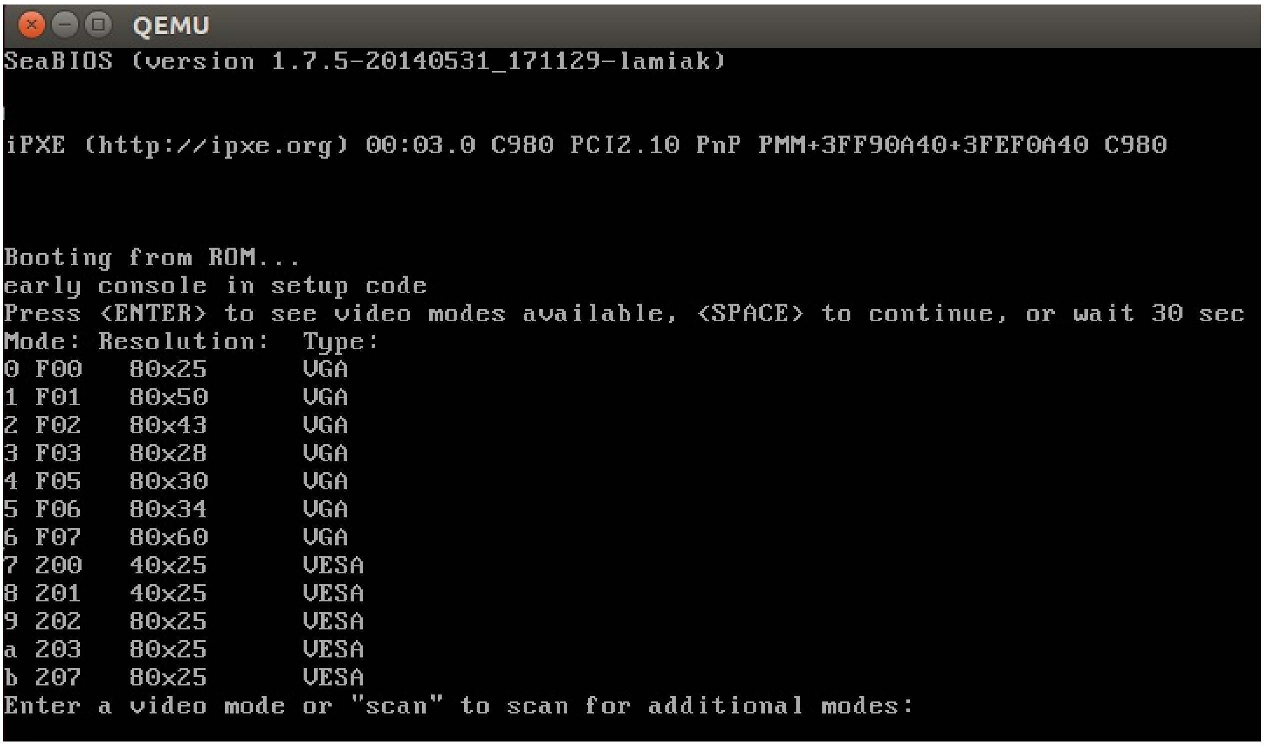
至于 `boot_params.hdr` 数据结构中的内容，是通过 `copy_boot_params` 函数实现的 （关于这个函数的实现细节请查看上一章的内容），`boot_params.hdr` 中的 `vid_mode` 是引导程序必须填入的字段。你可以在 `kernel boot protocol` 文档中找到关于 `vid_mode` 的详细信息：

```
Offset    Proto    Name        Meaning
/Size
01FA/2    ALL        vid_mode    Video mode control
```

而在 `linux kernel boot protocol` 文档中定义了如何通过命令行参数的方式为 `vid_mode` 字段传入相应的值：

```
**** SPECIAL COMMAND LINE OPTIONS
vga=<mode>
    <mode> here is either an integer (in C notation, either
    decimal, octal, or hexadecimal) or one of the strings
    "normal" (meaning 0xFFFF), "ext" (meaning 0xFFFE) or "ask"
    (meaning 0xFFFD).  This value should be entered into the
    vid_mode field, as it is used by the kernel before the command
    line is parsed.
```

根据上面的描述，我们可以通过将 `vga` 选项写入 grub 或者写到引导程序的配置文件，从而让内核命令行得到相应的显示模式设置信息。这个选项可以接受不同类型的值来表示相同的意思。比如你可以传入 0XFFFD 或者 ask，这2个值都表示需要显示一个菜单让用户选择想要的显示模式。下面的链接就给出了这个菜单：



通过这个菜单，用户可以选择想要进入的显示模式。不过在我们进一步了解显示模式的设置过程之前，让我们先回头了解一些重要的概念。

内核数据类型

在前面的章节中，我们已经接触到了一个类似于 `u16` 的内核数据类型。下面列出了更多内核支持的数据类型：

Type	char	short	int	long	u8	u16	u32	u64
Size	1	2	4	8	1	2	4	8

如果你尝试阅读内核代码，最好能够牢记这些数据类型。

堆操作 API

在 `set_video` 函数将 `vid_mod` 的值设置完成之后，将调用 `RESET_HEAP` 宏将 HEAP 头指向 `_end` 符号。`RESET_HEAP` 宏定义在 boot.h：

```
#define RESET_HEAP() ((void *)( HEAP = _end ))
```

如果你阅读过第二部分，你应该还记得在第二部分中，我们通过 `init_heap` 函数完成了 HEAP 的初始化。在 `boot.h` 中定义了一系列的方法来操作被初始化之后的 HEAP。这些操作包括：

```
#define RESET_HEAP() ((void *)( HEAP = _end ))
```

就像我们在前面看到的，这个宏只是简单的将 HEAP 头设置到 `_end` 标号。在上一章中我们已经说明了 `_end` 标号，在 `boot.h` 中通过 `extern char _end[];` 来引用（从这里可以看出，在内核初始化的时候堆和栈是共享内存空间的，详细的信息可以查看第一章的堆栈初始化和第二章的堆初始化）：

下面一个是 `GET_HEAP` 宏：

```
#define GET_HEAP(type, n) \
    ((type *)__get_heap(sizeof(type),__alignof__(type),(n)))
```

可以看出这个宏调用了 `__get_heap` 函数来进行内存的分配。`__get_heap` 需要下面3个参数来进行内存分配操作：

- 某个数据类型所占用的字节数
- `__alignof__(type)` 返回对于请求的数据类型需要怎样的对齐方式 (根据我的了解这个是 gcc 提供的一个功能)
- `n` 需要分配多少个对应数据类型的对象

下面是 `__get_heap` 函数的实现：

```
static inline char *__get_heap(size_t s, size_t a, size_t n)
{
    char *tmp;

    HEAP = (char *)(((size_t)HEAP+(a-1)) & ~(a-1));
    tmp = HEAP;
    HEAP += s*n;
    return tmp;
}
```

现在让我们来了解这个函数是如何工作的。 这个函数首先根据对齐方式要求（参数 `a` ）调整 `HEAP` 的值，然后将 `HEAP` 值赋值给一个临时变量 `tmp` 。接下来根据需要分配的对象的个数（参数 `n` ），预留出所需要的内存，然后将 `tmp` 返回给调用端。

最后一个关于 HEAP 的操作是：

```
static inline bool heap_free(size_t n)
{
    return (int)(heap_end - HEAP) >= (int)n;
}
```

这个函数简单做了一个减法 `heap_end - HEAP` ，如果相减的结果大于请求的内存，那么就返回真，否则返回假。

我们已经看到了所有可以对 HEAP 进行操作，下面让我们继续显示模式设置过程。

设置显示模式

在我们分析了内核数据类型以及和 HEAP 相关的操作之后，让我们回来继续分析显示模式的初始化。在 `RESET_HEAP()` 函数被调用之后， `set_video` 函数接着调用 `store_mode_params` 函数将对应显示模式的相关参数写入 `boot_params.screen_info` 字段。这个字段的结构定义可以在 include/uapi/linux/screen_info.h 中找到。

`store_mode_params` 函数将调用 `store_cursor_position` 函数将当前屏幕上光标的位置保存起来。下面让我们来看 `store_cursor_poistion` 函数是如何实现的。

首先函数初始化一个类型为 `biosregs` 的变量，将其中的 `AH` 寄存器内容设置成 `0x3` ，然后调用 `0x10` BIOS 中断。当中断调用返回之后， `DL` 和 `DH` 寄存器分别包含了当前光标的行和列信息。接着，这2个信息将被保存到 `boot_params.screen_info` 字段的 `orig_x` 和 `orig_y` 字段。

在 `store_cursor_position` 函数执行完毕之后， `store_mode_params` 函数将调用 `store_video_mode` 函数将当前使用的显示模式保存到 `boot_params.screen_info.orig_video_mode` 。

接下來 `store_mode_params` 函数将根据当前显示模式的设定，给 `video_segment` 变量设置正确的值（实际上就是设置显示内存的起始地址）。在 BIOS 将控制权转移到引导扇区的时候，显示内存地址和显示模式的对应关系如下表所示：

```
0xB000:0x0000     32 Kb     Monochrome Text Video Memory
0xB800:0x0000     32 Kb     Color Text Video Memory
```

根据上表，如果当前显示模式是 MDA, HGC 或者单色 VGA 模式，那么 `video_sgement` 的值将被设置成 `0xB000` ；如果当前显示模式是彩色模式，那么 `video_segment` 的值将被设置成 `0xB800` 。在这之后， `store_mode_params` 函数将保存字体大小信息到 `boot_params.screen_info.orig_video_points` ：

```
//保存字体大小信息
set_fs(0);
font_size = rdfs16(0x485);
boot_params.screen_info.orig_video_points = font_size;
```

这段代码首先调用 `set_fs` 函数（在 boot.h 中定义了许多类似的函数进行寄存器操作）将数字 `0` 放入 `FS` 寄存器。接着从内存地址 `0x485` 处获取字体大小信息并保存到 `boot_params.screen_info.orig_video_points` 。

```
  x = rdfs16(0x44a);
  y = (adapter == ADAPTER_CGA) ? 25 : rdfs8(0x484)+1;
```

接下来代码将从地址 `0x44a` 处获得屏幕列信息，从地址 `0x484` 处获得屏幕行信息，并将它们保存到 `boot_params.screen_info.orig_video_cols` 和 `boot_params.screen_info.orig_video_lines` 。到这里， `store_mode_params` 的执行就结束了。

接下来， `set_video` 函数将调用 `save_screen` 函数将当前屏幕上的所有信息保存到 HEAP 中。这个函数首先获得当前屏幕的所有信息（包括屏幕大小，当前光标位置，屏幕上的字符信息），并且保存到 `saved_screen` 结构体中。这个结构体的定义如下所示：

```
static struct saved_screen {
    int x, y;
    int curx, cury;
    u16 *data;
} saved;
```

接下来函数将检查 HEAP 中是否有足够的空间保存这个结构体的数据：

```
if (!heap_free(saved.x*saved.y*sizeof(u16)+512))
        return;
```

如果 HEAP 有足够的空间，代码将在 HEAP 中分配相应的空间并且将 `saved_screen` 保存到 HEAP。

接下来 `set_video` 函数将调用 `probe_cards(0)` （这个函数定义在 arch/x86/boot/video-mode.c）。这个函数简单遍历所有的显卡，并通过调用驱动程序设置显卡所支持的显示模式：

```
for (card = video_cards; card < video_cards_end; card++) {
        if (card->unsafe == unsafe) {
            if (card->probe)
                card->nmodes = card->probe();
            else
                card->nmodes = 0;
        }
}
```

如果你仔细看上面的代码，你会发现 `video_cards` 这个变量并没有被声明，那么程序怎么能够正常编译执行呢？实际上很简单，它指向了一个在 arch/x86/boot/setup.ld 中定义的叫做 `.videocards` 的内存段：

```
    .videocards    : {
        video_cards = .;
        *(.videocards)
        video_cards_end = .;
    }
```

那么这段内存里面存放的数据是什么呢，下面我们就来详细分析。在内核初始化代码中，对于每个支持的显示模式都是使用下面的代码进行定义的：

```
static __videocard video_vga = {
    .card_name    = "VGA",
    .probe        = vga_probe,
    .set_mode    = vga_set_mode,
};
```

`__videocard` 是一个宏定义，如下所示：

```
#define __videocard struct card_info __attribute__((used,section(".videocards")))
```

因此 `__videocard` 是一个 `card_info` 结构，这个结构定义如下：

```
struct card_info {
    const char *card_name;
    int (*set_mode)(struct mode_info *mode);
    int (*probe)(void);
    struct mode_info *modes;
    int nmodes;
    int unsafe;
    u16 xmode_first;
    u16 xmode_n;
};
```

在 `.videocards` 内存段实际上存放的就是所有被内核初始化代码定义的 `card_info` 结构（可以看成是一个数组），所以 `probe_cards` 函数可以使用 `video_cards`，通过循环遍历所有的 `card_info` 。

在 `probe_cards` 执行完成之后，我们终于进入 `set_video` 函数的主循环了。在这个循环中，如果 `vid_mode=ask` ，那么将显示一个菜单让用户选择想要的显示模式，然后代码将根据用户的选择或者 `vid_mod` 的值 ，通过调用 `set_mode` 函数来设置正确的显示模式。如果设置成功，循环结束，否则显示菜单让用户选择显示模式，继续进行设置显示模式的尝试。

```
for (;;) {
        if (mode == ASK_VGA)
            mode = mode_menu();

        if (!set_mode(mode))
            break;

        printf("Undefined video mode number: %x\n", mode);
        mode = ASK_VGA;
    }
```

你可以在 video-mode.c 中找到 `set_mode` 函数的定义。这个函数只接受一个参数，这个参数是对应的显示模式的数字表示（这个数字来自于显示模式选择菜单，或者从内核命令行参数获得）。

`set_mode` 函数首先检查传入的 `mode` 参数，然后调用 `raw_set_mode` 函数。而后者将遍历内核知道的所有 `card_info` 信息，如果发现某张显卡支持传入的模式，这调用 `card_info` 结构中保存的 `set_mode` 函数地址进行显卡显示模式的设置。以 `video_vga` 这个 `card_info` 结构来说，保存在其中的 `set_mode` 函数就指向了 `vga_set_mode` 函数。下面的代码就是 `vga_set_mode` 函数的实现，这个函数根据输入的 vga 显示模式，调用不同的函数完成显示模式的设置：

```
static int vga_set_mode(struct mode_info *mode)
{
    vga_set_basic_mode();

    force_x = mode->x;
    force_y = mode->y;

    switch (mode->mode) {
    case VIDEO_80x25:
        break;
    case VIDEO_8POINT:
        vga_set_8font();
        break;
    case VIDEO_80x43:
        vga_set_80x43();
        break;
    case VIDEO_80x28:
        vga_set_14font();
        break;
    case VIDEO_80x30:
        vga_set_80x30();
        break;
    case VIDEO_80x34:
        vga_set_80x34();
        break;
    case VIDEO_80x60:
        vga_set_80x60();
        break;
    }
    return 0;
}
```

在上面的代码中，每个 `vga_set***` 函数只是简单调用 `0x10` BIOS 中断来进行显示模式的设置。

在显卡的显示模式被正确设置之后，这个最终的显示模式被写回 `boot_params.hdr.vid_mode` 。

接下来 `set_video` 函数将调用 `vesa_store_edid` 函数， 这个函数只是简单的将 EDID (**E**xtended **D**isplay **I**dentification **D**ata) 写入内存，以便于内核访问。最后， `set_video` 将调用 `do_restore` 函数将前面保存的当前屏幕信息还原到屏幕上。

到这里为止，显示模式的设置完成，接下来我们可以切换到保护模式了。

在切换到保护模式之前的最后的准备工作

在进入保护模式之前的最后一个函数调用发生在 main.c 中的 `go_to_protected_mode` 函数，就像这个函数的注释说的，这个函数将进行最后的准备工作然后进入保护模式，下面就让我们来具体看看最后的准备工作是什么，以及系统是如何切换到保护模式的。

`go_to_protected_mode` 函数本身定义在 arch/x86/boot/pm.c。 这个函数调用了一些其他的函数进行最后的准备工作，下面就让我们来具体看看这些函数。

`go_to_protected_mode` 函数首先调用的是 `realmode_switch_hook` 函数，后者如果发现 `realmode_switch` hook， 那么将调用它并禁止 NMI 中断，反之将直接禁止 NMI 中断。只有当 bootloader 运行在宿主环境下（比如在 DOS 下运行 ）， hook 才会被使用。你可以在 boot protocol (see **ADVANCED BOOT LOADER HOOKS**) 中详细了解 hook 函数的信息。

```
/*
 * Invoke the realmode switch hook if present; otherwise
 * disable all interrupts.
 */
static void realmode_switch_hook(void)
{
    if (boot_params.hdr.realmode_swtch) {
        asm volatile("lcallw *%0"
                 : : "m" (boot_params.hdr.realmode_swtch)
                 : "eax", "ebx", "ecx", "edx");
    } else {
        asm volatile("cli");
        outb(0x80, 0x70); /* Disable NMI */
        io_delay();
    }
}
```

`realmode_switch` 指向了一个16 位实模式代码地址（远跳转指针），这个16位代码将禁止 NMI 中断。所以在上述代码中，如果 `realmode_swtch` hook 存在，代码是用了 `lcallw` 指令进行远函数调用。在我的环境中，因为不存在这个 hook ，所以代码是直接进入 `else` 部分进行了 NMI 的禁止：

```
asm volatile("cli");
outb(0x80, 0x70);    /* Disable NMI */
io_delay();
```

上面的代码首先调用 `cli` 汇编指令清除了中断标志 `IF` ，这条指令执行之后，外部中断就被禁止了，紧接着的下一行代码就禁止了 NMI 中断。

这里简单介绍一下中断。中断是由硬件或者软件产生的，当中断产生的时候， CPU 将得到通知。这个时候， CPU 将停止当前指令的执行，保存当前代码的环境，然后将控制权移交到中断处理程序。当中断处理程序完成之后，将恢复中断之前的运行环境，从而被中断的代码将

继续运行。 NMI 中断是一类特殊的中断，往往预示着系统发生了不可恢复的错误，所以在正常运行的操作系统中，NMI 中断是不会被禁止的，但是在进入保护模式之前，由于特殊需求，代码禁止了这类中断。我们将在后续的章节中对中断做更多的介绍，这里就不展开了。

现在让我们回到上面的代码，在 NMI 中断被禁止之后（通过写 `0x80` 进 CMOS 地址寄存器 `0x70` ），函数接着调用了 `io_delay` 函数进行了短暂的延时以等待 I/O 操作完成。下面就是 `io_delay` 函数的实现：

```
static inline void io_delay(void)
{
    const u16 DELAY_PORT = 0x80;
    asm volatile("outb %%al,%0" : : "dN" (DELAY_PORT));
}
```

对 I/O 端口 `0x80` 写入任何的字节都将得到 1 ms 的延时。在上面的代码中，代码将 `al` 寄存器中的值写到了这个端口。在这个 `io_delay` 调用完成之后， `realmode_switch_hook` 函数就完成了所有工作，下面让我们进入下一个函数。

下一个函数调用是 `enable_a20` ，这个函数使能 A20 line，你可以在 arch/x86/boot/a20.c 找到这个函数的定义，这个函数会尝试使用不同的方式来使能 A20 地址线。首先这个函数将调用 `a20_test_short` （该函数将调用 `a20_test` 函数） 来检测 A20 地址线是否已经被激活了：

```
static int a20_test(int loops)
{
    int ok = 0;
    int saved, ctr;

    set_fs(0x0000);
    set_gs(0xffff);

    saved = ctr = rdfs32(A20_TEST_ADDR);

    while (loops--) {
        wrfs32(++ctr, A20_TEST_ADDR);
        io_delay();    /* Serialize and make delay constant */
        ok = rdgs32(A20_TEST_ADDR+0x10) ^ ctr;
        if (ok)
            break;
    }

    wrfs32(saved, A20_TEST_ADDR);
    return ok;
}
```

这个函数首先将 `0x0000` 放入 `FS` 寄存器，将 `0xffff` 放入 `GS` 寄存器。然后通过 `rdfs32` 函数调用，将 `A20_TEST_ADDR` 内存地址的内容放入 `saved` 和 `ctr` 变量。

接下来我们使用 `wrfs32` 函数将更新过的 `ctr` 的值写入 `fs:gs` ，接着延时 1ms， 然后从 `GS:A20_TEST_ADDR+0x10` 读取内容，如果该地址内容不为0，那么 A20 已经被激活。如果 A20 没有被激活，代码将尝试使用多种方法进行 A20 地址激活。其中的一种方法就是调用 BIOS `0X15` 中断激活 A20 地址线。

如果 `enabled_a20` 函数调用失败，显示一个错误消息并且调用 `die` 函数结束操作系统运行。`die` 函数定义在 arch/x86/boot/header.S:

```
die:
    hlt
    jmp     die
    .size    die, .-die
```

A20 地址线被激活之后，`reset_coprocessor` 函数被调用：

```
outb(0, 0xf0);
outb(0, 0xf1);
```

这个函数非常简单，通过将 `0` 写入 I/O 端口 `0xf0` 和 `0xf1` 以复位数字协处理器。

接下来 `mask_all_interrupts` 函数将被调用：

```
outb(0xff, 0xa1);       /* Mask all interrupts on the secondary PIC */
outb(0xfb, 0x21);       /* Mask all but cascade on the primary PIC */
```

这个函数调用屏蔽了从中断控制器 (注：中断控制器的原文是 Programmable Interrupt Controller) 的所有中断，和主中断控制器上除IRQ2以外的所有中断（IRQ2是主中断控制器上的级联中断，所有从中断控制器的中断将通过这个级联中断报告给 CPU ）。

到这里位置，我们就完成了所有的准备工作，下面我们就将正式开始从实模式转换到保护模式。

设置中断描述符表

现在内核将调用 `setup_idt` 方法来设置中断描述符表（ IDT ）：

```
static void setup_idt(void)
{
    static const struct gdt_ptr null_idt = {0, 0};
    asm volatile("lidtl %0" : : "m" (null_idt));
}
```

上面的代码使用 `lidtl` 指令将 `null_idt` 所指向的中断描述符表引入寄存器 IDT。由于 `null_idt` 没有设定中断描述符表的长度（长度为 0 ），所以这段指令执行之后，实际上没有任何中断调用被设置成功（所有中断调用都是空的），在后面的章节中我们将看到正确的设置。 `null_idt` 是一个 `gdt_ptr` 结构的数据，这个结构的定义如下所示：

```
struct gdt_ptr {
    u16 len;
    u32 ptr;
} __attribute__((packed));
```

在上面的定义中，我们可以看到上面这个结构包含一个 16 bit 的长度字段，和一个 32 bit 的指针字段。 `__attribute__((packed))` 意味着这个结构就只包含 48 bit 信息（没有字节对齐优化）。在下面一节中，我们将看到相同的结构将被导入 `GDTR` 寄存器（如果你还记得上一章的内容，应该记得 GDTR 寄存器是 48 bit 长度的）。

设置全局描述符表

在设置完中断描述符表之后，我们将使用 `setup_gdt` 函数来设置全局描述符表（关于全局描述符表，大家可以参考上一章 的内容）。在 `setup_gdt` 函数中，使用 `boot_gdt` 数组定义了需要引入 GDTR 寄存器的段描述符信息：

```
   //GDT_ENTRY_BOOT_CS 定义在http://lxr.free-electrons.com/source/arch/x86/include/asm/segment.h#L19 = 2
    static const u64 boot_gdt[] __attribute__((aligned(16))) = {
        [GDT_ENTRY_BOOT_CS] = GDT_ENTRY(0xc09b, 0, 0xfffff),
        [GDT_ENTRY_BOOT_DS] = GDT_ENTRY(0xc093, 0, 0xfffff),
        [GDT_ENTRY_BOOT_TSS] = GDT_ENTRY(0x0089, 4096, 103),
    };
```

在上面的 `boot_gdt` 数组中，我们定义了代码，数据和 TSS 段(Task State Segment, 任务状态段)的段描述符，因为我们并没有设置任何的中断调用（记得上面说的 `null_idt` 吗？），所以 TSS 段并不会被使用到。TSS 段存在的唯一目的就是让 Intel 处理器能够正确进入保护模式。下面让我们详细了解一下 `boot_gdt` 这个数组，首先，这个数组被 `__attribute__((aligned(16)))` 修饰，这就意味着这个数组将以 16 字节为单位对齐。让我们通过下面的例子来了解一下什么叫 16 字节对齐：

```
#include <stdio.h>

struct aligned {
    int a;
}__attribute__((aligned(16)));

struct nonaligned {
    int b;
};

int main(void)
{
    struct aligned    a;
    struct nonaligned na;

    printf("Not aligned - %zu \n", sizeof(na));
    printf("Aligned - %zu \n", sizeof(a));

    return 0;
}
```

上面的代码可以看出，一旦指定了 16 字节对齐，即使结构中只有一个 `int` 类型的字段，整个结构也将占用 16 个字节：

```
$ gcc test.c -o test && test
Not aligned - 4
Aligned - 16
```

因为在 `boot_gdt` 的定义中， `GDT_ENTRY_BOOT_CS = 2` ，所以在数组中有2个空项，第一项是一个空的描述符，第二项在代码中没有使用。在没有 `align 16` 之前，整个结构占用了（8*5=40）个字节，加了 `align 16` 之后，结构就占用了 48 字节 。

上面代码中出现的 `GDT_ENTRY` 是一个宏定义，这个宏接受 3 个参数（标志，基地址，段长度）来产生段描述符结构。让我们来具体分析上面数组中的代码段描述符（ `GDT_ENTRY_BOOT_CS` ）来看看这个宏是如何工作的，对于这个段， `GDT_ENTRY` 接受了下面 3 个参数：

- 基地址 - 0
- 段长度 - 0xfffff
- 标志 - 0xc09b

上面这些数字表明，这个段的基地址是 0， 段长度是 `0xfffff` （ 1 MB ） ，而标志字段展开之后是下面的二进制数据：

```
1100 0000 1001 1011
```

这些二进制数据的具体含义如下:

- 1 - (G) 这里为 1，表示段的实际长度是 `0xfffff * 4kb` = `4GB`
- 1 - (D) 表示这个段是一个32位段
- 0 - (L) 这个代码段没有运行在 long mode
- 0 - (AVL) Linux 没有使用
- 0000 - 段长度的4个位
- 1 - (P) 段已经位于内存中
- 00 - (DPL) - 段优先级为0
- 1 - (S) 说明这个段是一个代码或者数据段
- 101 - 段类型为可执行/可读
- 1 - 段可访问

关于段描述符的更详细的信息你可以从上一章中获得 上一章，你也可以阅读 Intel® 64 and IA-32 Architectures Software Developer's Manuals 3A获取全部信息。

在定义了数组之后，代码将获取 GDT 的长度：

```
gdt.len = sizeof(boot_gdt)-1;
```

接下来是将 GDT 的地址放入 gdt.ptr 中：

```
gdt.ptr = (u32)&boot_gdt + (ds() << 4);
```

这里的地址计算很简单，因为我们还在实模式，所以就是 （ ds << 4 + 数组起始地址）。

最后通过执行 `lgdtl` 指令将 GDT 信息写入 GDTR 寄存器：

```
asm volatile("lgdtl %0" : : "m" (gdt));
```

切换进入保护模式

`go_to_protected_mode` 函数在完成 IDT, GDT 初始化，并禁止了 NMI 中断之后，将调用 `protected_mode_jump` 函数完成从实模式到保护模式的跳转：

```
protected_mode_jump(boot_params.hdr.code32_start, (u32)&boot_params + (ds() << 4));
```

`protected_mode_jump` 函数定义在 arch/x86/boot/pmjump.S，它接受下面2个参数:

- 保护模式代码的入口
- `boot_params` 结构的地址

第一个参数保存在 `eax` 寄存器，而第二个参数保存在 `edx` 寄存器。

代码首先在 `boot_params` 地址放入 `esi` 寄存器，然后将 `cs` 寄存器内容放入 `bx` 寄存器，接着执行 `bx << 4 + 标号为2的代码的地址`，这样一来 `bx` 寄存器就包含了标号为2的代码的地址。接下来代码将把数据段索引放入 `cx` 寄存器，将 TSS 段索引放入 `di` 寄存器：

```
movw    $__BOOT_DS, %cx
movw    $__BOOT_TSS, %di
```

就像前面我们看到的 `GDT_ENTRY_BOOT_CS` 的值为2，每个段描述符都是 8 字节，所以 `cx` 寄存器的值将是 `2*8 = 16`，`di` 寄存器的值将是 `4*8 =32`。

接下来，我们通过设置 `CR0` 寄存器相应的位使 CPU 进入保护模式：

```
movl    %cr0, %edx
orb    $X86_CR0_PE, %dl
movl    %edx, %cr0
```

在进入保护模式之后，通过一个长跳转进入 32 位代码：

```
    .byte    0x66, 0xea
2:    .long    in_pm32
    .word    __BOOT_CS ;(GDT_ENTRY_BOOT_CS*8) = 16，段描述符表索引
```

这段代码中

- `0x66` 操作符前缀允许我们混合执行 16 位和 32 位代码
- `0xea` - 跳转指令的操作符
- `in_pm32` 跳转地址偏移
- `__BOOT_CS` 代码段描述符索引

在执行了这个跳转命令之后，我们就在保护模式下执行代码了：

```
.code32
.section ".text32","ax"
```

保护模式代码的第一步就是重置所有的段寄存器（除了 `CS` 寄存器）：

```
GLOBAL(in_pm32)
movl    %ecx, %ds
movl    %ecx, %es
movl    %ecx, %fs
movl    %ecx, %gs
movl    %ecx, %ss
```

还记得我们在实模式代码中将 `$__BOOT_DS` （数据段描述符索引）放入了 `cx` 寄存器，所以上面的代码设置所有段寄存器（除了 `cs` 寄存器）指向数据段。接下来代码将所有的通用寄存器清 0 ：

```
xorl    %ecx, %ecx
xorl    %edx, %edx
xorl    %ebx, %ebx
xorl    %ebp, %ebp
xorl    %edi, %edi
```

最后使用长跳转跳入正在的 32 位代码（通过参数传入的地址）

```
jmpl    *%eax ;?jmpl cs:eax?
```

到这里，我们就进入了保护模式开始执行代码了，下一章我们将分析这段 32 位代码到底做了些什么。

结论

这章到这里就结束了，在下一章中我们将具体介绍这章最后跳转到的 32 位代码，并且了解系统是如何进入 long mode的。

如果你有任何的问题或者建议，你可以留言，也可以直接发消息给我twitter.

如果你发现文中描述有任何问题，请提交一个 **PR** 到 **linux-insides-zh** 。

链接

- VGA
- VESA BIOS Extensions
- Data structure alignment
- Non-maskable interrupt
- A20
- GCC designated inits
- GCC type attributes
- Previous part

内核引导过程. Part 4.

切换到64位模式

这是 `内核引导过程` 的第四部分，我们将会看到在保护模式中的最初几步，比如确认CPU是否支持长模式，SSE和分页以及页表的初始化，在这部分的最后我们还将讨论如何切换到长模式。

注意：这部分将会有大量的汇编代码，如果你不熟悉汇编，建议你找本书参考一下。

在前一章节，我们停在了跳转到位于 arch/x86/boot/pmjump.S 的 32 位入口点这一步：

```
jmpl    *%eax
```

回忆一下， `eax` 寄存器包含了 32 位入口点的地址。我们可以在 x86 linux 内核引导协议 中找到相关内容：

```
When using bzImage, the protected-mode kernel was relocated to 0x100000
```

```
当使用 bzImage 时，保护模式下的内核被重定位至 0x100000
```

让我们检查一下 32 位入口点的寄存器值来确保这是对的：

```
eax            0x100000    1048576
ecx            0x0         0
edx            0x0         0
ebx            0x0         0
esp            0x1ff5c     0x1ff5c
ebp            0x0         0x0
esi            0x14470     83056
edi            0x0         0
eip            0x100000    0x100000
eflags         0x46        [ PF ZF ]
cs             0x10    16
ss             0x18    24
ds             0x18    24
es             0x18    24
fs             0x18    24
gs             0x18    24
```

我们在这里可以看到 `cs` 寄存器包含了 - `0x10` （回忆前一章节，这代表了全局描述符表中的第二个索引项）， `eip` 寄存器的值是 `0x100000`，并且包括代码段在内的所有内存段的基地址都为0。所以我们可以得到物理地址： `0:0x100000` 或者 `0x100000`，这和协议规定的一样。现在让我们从 32 位入口点开始。

32 位入口点

我们可以在汇编源码 arch/x86/boot/compressed/head_64.S 中找到 32 位入口点的定义。

```
    __HEAD
    .code32
ENTRY(startup_32)
....
....
....
ENDPROC(startup_32)
```

首先，为什么目录名叫做 `被压缩的 (compressed)` ？实际上 `bzimage` 是由 `vmlinux + 头文件 + 内核启动代码` 被 gzip 压缩之后获得的。我们在前几个章节已经看到了启动内核的代码。所以， `head_64.S` 的主要目的就是做好进入长模式的准备之后进入长模式，进入以后再解压内核。在这一章节，我们将会看到直到内核解压缩之前的所有步骤。

在 `arch/x86/boot/compressed` 目录下有两个文件：

- head_32.S
- head_64.S

但是，你可能还记得我们这本书只和 `x86_64` 有关，所以我们只会关注 `head_64.S` ；在我们这里 `head_32.S` 没有被用到。让我们看一下 arch/x86/boot/compressed/Makefile。在那里我们可以看到以下目标：

```
vmlinux-objs-y := $(obj)/vmlinux.lds $(obj)/head_$(BITS).o $(obj)/misc.o \
    $(obj)/string.o $(obj)/cmdline.o \
    $(obj)/piggy.o $(obj)/cpuflags.o
```

注意 `$(obj)/head_$(BITS).o` 。这意味着我们将会选择基于 `$(BITS)` 所设置的文件执行链接操作，即 head_32.o 或者 head_64.o。 `$(BITS)` 在 arch/x86/Makefile 之中根据 .config 文件另外定义：

```
ifeq ($(CONFIG_X86_32),y)
        BITS := 32
        ...
        ...
else
        BITS := 64
        ...
        ...
endif
```

现在我们知道从哪里开始了，那就来吧。

必要时重新加载内存段寄存器

正如上面阐述的，我们先从 arch/x86/boot/compressed/head_64.S 这个汇编文件开始。首先我们看到了在 `startup_32` 之前的特殊段属性定义：

```
    __HEAD
    .code32
ENTRY(startup_32)
```

这个 `__HEAD` 是一个定义在头文件 include/linux/init.h 中的宏，展开后就是下面这个段的定义：

```
#define __HEAD        .section    ".head.text","ax"
```

其拥有 `.head.text` 的命名和 `ax` 标记。在这里，这些标记告诉我们这个段是可执行的或者换种说法，包含了代码。我们可以在 arch/x86/boot/compressed/vmlinux.lds.S 这个链接脚本里找到这个段的定义：

```
SECTIONS
{
    . = 0;
    .head.text : {
        _head = . ;
        HEAD_TEXT
        _ehead = . ;
    }
```

如果你不熟悉 `GNU LD` 这个链接脚本语言的语法，你可以在这个文档中找到更多信息。简单来说，这个 `.` 符号是一个链接器的特殊变量 - 位置计数器。其被赋值为相对于该段的偏移。在这里，我们将位置计数器赋值为0，这意味着我们的代码被链接到内存的 `0` 偏移处。此

外，我们可以从注释里找到更多信息：

```
Be careful parts of head_64.S assume startup_32 is at address 0.
```

```
要小心， head_64.S 中一些部分假设 startup_32 位于地址 0。
```

好了，现在我们知道我们在哪里了，接下来就是深入 `startup_32` 函数的最佳时机。

在 `startup_32` 函数的开始，我们可以看到 `cld` 指令将标志寄存器的 `DF` （方向标志）位清空。当方向标志被清空，所有的串操作指令像stos， scas等等将会增加索引寄存器 `esi` 或者 `edi` 的值。我们需要清空方向标志是因为接下来我们会使用汇编的串操作指令来做为页表腾出空间等工作。

在我们清空 `DF` 标志后，下一步就是从内核加载头中的 `loadflags` 字段来检查 `KEEP_SEGMENTS` 标志。你是否还记得在本书的最初一节，我们已经看到过 `loadflags` 。在那里我们检查了 `CAN_USE_HEAP` 标记以使用堆。现在我们需要检查 `KEEP_SEGMENTS` 标记。这些标记在 linux 的引导协议文档中有描述：

```
Bit 6 (write): KEEP_SEGMENTS
  Protocol: 2.07+
  - If 0, reload the segment registers in the 32bit entry point.
  - If 1, do not reload the segment registers in the 32bit entry point.
    Assume that %cs %ds %ss %es are all set to flat segments with
    a base of 0 (or the equivalent for their environment).
```

```
第 6 位 (写): KEEP_SEGMENTS
  协议版本: 2.07+
  - 为0，在32位入口点重载段寄存器
  - 为1，不在32位入口点重载段寄存器。假设 %cs %ds %ss %es 都被设到基地址为0的普通段中（或者在他们的环境中等价的位置）。
```

所以，如果 `KEEP_SEGMENTS` 位在 `loadflags` 中没有被设置，我们需要重置 `ds` , `ss` 和 `es` 段寄存器到一个基地址为 `0` 的普通段中。如下：

```
    testb $(1 << 6), BP_loadflags(%esi)
    jnz 1f

    cli
    movl    $(__BOOT_DS), %eax
    movl    %eax, %ds
    movl    %eax, %es
    movl    %eax, %ss
```

记住 `__BOOT_DS` 是 `0x18` （位于全局描述符表中数据段的索引）。如果设置了 `KEEP_SEGMENTS` ，我们就跳转到最近的 `1f` 标签，或者当没有 `1f` 标签，则用 `__BOOT_DS` 更新段寄存器。这非常简单，但是这是一个有趣的操作。如果你已经读了前一章节，你或许还记得我们在 arch/x86/boot/pmjump.S 中切换到保护模式的时候已经更新了这些段寄存器。那么为什么我们还要去关心这些段寄存器的值呢？答案很简单，Linux 内核也有32位的引导协议，如果一个引导程序之前使用32位协议引导内核，那么在 `startup_32` 之前的代码就会被忽略。在这种情况下 `startup_32` 将会变成引导程序之后的第一个入口点，不保证段寄存器会不会处于未知状态。

在我们检查了 `KEEP_SEGMENTS` 标记并且给段寄存器设置了正确的值之后，下一步就是计算我们代码的加载和编译运行之间的位置偏差了。记住 `setup.ld.S` 包含了以下定义：在 `.head.text` 段的开始 `. = 0` 。这意味着这一段代码被编译成从 `0` 地址运行。我们可以在 `objdump` 工具的输出中看到：

```
arch/x86/boot/compressed/vmlinux:     file format elf64-x86-64


Disassembly of section .head.text:

0000000000000000 <startup_32>:
   0:   fc                      cld
   1:   f6 86 11 02 00 00 40    testb  $0x40,0x211(%rsi)
```

`objdump` 工具告诉我们 `startup_32` 的地址是 `0` 。但实际上并不是。我们当前的目标是获知我们实际上在哪里。在长模式下，这非常简单，因为其支持 `rip` 相对寻址，但是我们当前处于保护模式下。我们将会使用一个常用的方法来确定 `startup_32` 的地址。我们需要定义一个标签并且跳转到它，然后把栈顶抛出到一个寄存器中：

```
call label
label: pop %reg
```

在这之后，那个寄存器将会包含标签的地址，让我们看看在 Linux 内核中类似的寻找 `startup_32` 地址的代码：

```
    leal    (BP_scratch+4)(%esi), %esp
    call    1f
1:  popl    %ebp
    subl    $1b, %ebp
```

回忆前一节， `esi` 寄存器包含了 boot_params 结构的地址，这个结构在我们切换到保护模式之前已经被填充了。 `bootparams` 这个结构体包含了一个特殊的字段 `scratch` ，其偏移量为 `0x1e4` 。这个 4 字节的区域将会成为 `call` 指令的临时栈。我们把 `scratch` 的地址加 4 存入 `esp` 寄存器。我们之所以在 `BP_scratch` 基础上加 `4` 是因为，如之前所说的，这将成

为一个临时的栈，而在 `x86_64` 架构下，栈是自顶向下生长的。所以我们的栈指针就会指向栈顶。接下来我们就可以看到我上面描述的过程。我们跳转到 `1f` 标签并且把该标签的地址放入 `ebp` 寄存器，因为在执行 `call` 指令之后我们把返回地址放到了栈顶。那么，目前我们拥有 `1f` 标签的地址，也能够很容易得到 `startup_32` 的地址。我们只需要把我们从栈里得到的地址减去标签的地址：

```
startup_32 (0x0)     +-----------------------+
                     |                       |
                     |                       |
                     |                       |
                     |                       |
                     |                       |
                     |                       |
                     |                       |
                     |                       |
1f (0x0 + 1f offset) +-----------------------+ %ebp - 实际物理地址
                     |                       |
                     |                       |
                     +-----------------------+
```

`startup_32` 被链接为在 `0x0` 地址运行，这意味着 `1f` 的地址为 `0x0 + 1f 的偏移量` 。实际上偏移量大概是 `0x22` 字节。 `ebp` 寄存器包含了 `1f` 标签的实际物理地址。所以如果我们从 `ebp` 中减去 `1f` ，我们就会得到 `startup_32` 的实际物理地址。Linux 内核的引导协议描述了保护模式下的内核基地址是 `0x100000` 。我们可以用 gdb 来验证。让我们启动调试器并且在 `1f` 的地址 `0x100022` 添加断点。如果这是正确的，我们将会看到在 `ebp` 寄存器中值为 `0x100022` ：

```
$ gdb
(gdb)$ target remote :1234
Remote debugging using :1234
0x0000fff0 in ?? ()
(gdb)$ br *0x100022
Breakpoint 1 at 0x100022
(gdb)$ c
Continuing.

Breakpoint 1, 0x00100022 in ?? ()
(gdb)$ i r
eax            0x18     0x18
ecx            0x0    0x0
edx            0x0    0x0
ebx            0x0    0x0
esp            0x144a8    0x144a8
ebp            0x100021    0x100021
esi            0x142c0    0x142c0
edi            0x0    0x0
eip            0x100022    0x100022
eflags         0x46     [ PF ZF ]
cs             0x10     0x10
ss             0x18     0x18
ds             0x18     0x18
es             0x18     0x18
fs             0x18     0x18
gs             0x18     0x18
```

如果我们执行下一条指令 `subl $1b, %ebp` ，我们将会看到：

```
nexti
...
ebp            0x100000    0x100000
...
```

好了，那是对的。`startup_32` 的地址是 `0x100000` 。在我们知道了 `startup_32` 的地址之后，我们可以开始准备切换到长模式了。我们的下一个目标是建立栈并且确认 CPU 对长模式和 SSE 的支持。

栈的建立和 CPU 的确认

如果不知道 `startup_32` 标签的地址，我们就无法建立栈。我们可以把栈看作是一个数组，并且栈指针寄存器 `esp` 必须指向数组的底部。当然我们可以在自己的代码里定义一个数组，但是我们需要知道其真实地址来正确配置栈指针。让我们看一下代码：

```
    movl    $boot_stack_end, %eax
    addl    %ebp, %eax
    movl    %eax, %esp
```

`boots_stack_end` 标签被定义在同一个汇编文件 arch/x86/boot/compressed/head_64.S 中，位于 .bss 段：

```
    .bss
    .balign 4
boot_heap:
    .fill BOOT_HEAP_SIZE, 1, 0
boot_stack:
    .fill BOOT_STACK_SIZE, 1, 0
boot_stack_end:
```

首先，我们把 `boot_stack_end` 放到 `eax` 寄存器中。那么 `eax` 寄存器将包含 `boot_stack_end` 链接后的地址或者说 `0x0 + boot_stack_end` 。为了得到 `boot_stack_end` 的实际地址，我们需要加上 `startup_32` 的实际地址。回忆一下，前面我们找到了这个地址并且把它存到了 `ebp` 寄存器中。最后，`eax` 寄存器将会包含 `boot_stack_end` 的实际地址，我们只需要将其加到栈指针上。

在外面建立了栈之后，下一步是 CPU 的确认。既然我们将要切换到 `长模式` ，我们需要检查 CPU 是否支持 `长模式` 和 `SSE` 。我们将会在跳转到 `verify_cpu` 函数之后执行：

```
    call    verify_cpu
    testl    %eax, %eax
    jnz    no_longmode
```

这个函数定义在 arch/x86/kernel/verify_cpu.S 中，只是包含了几个对 cpuid 指令的调用。该指令用于获取处理器的信息。在我们的情况下，它检查了对 `长模式` 和 `SSE` 的支持，通过 `eax` 寄存器返回0表示成功，1表示失败。

如果 `eax` 的值不是 0 ，我们就跳转到 `no_longmode` 标签，用 `hlt` 指令停止 CPU ，期间不会发生硬件中断：

```
no_longmode:
1:
    hlt
    jmp    1b
```

如果 `eax` 的值为0，万事大吉，我们可以继续。

计算重定位地址

下一步是在必要的时候计算解压缩之后的地址。首先，我们需要知道内核重定位的意义。我们已经知道 Linux 内核的32位入口点地址位于 `0x100000` 。但是那是一个32位的入口。默认的内核基地址由内核配置项 `CONFIG_PHYSICAL_START` 的值所确定，其默认值为 `0x1000000` 或 `16 MB` 。这里的主要问题是如果内核崩溃了，内核开发者需要一个配置于不同地址加载的 `救援内核` 来进行 kdump。Linux 内核提供了特殊的配置选项以解决此问题 - `CONFIG_RELOCATABLE` 。我们可以在内核文档中找到：

```
This builds a kernel image that retains relocation information
so it can be loaded someplace besides the default 1MB.

Note: If CONFIG_RELOCATABLE=y, then the kernel runs from the address
it has been loaded at and the compile time physical address
(CONFIG_PHYSICAL_START) is used as the minimum location.
```

```
这建立了一个保留了重定向信息的内核镜像，这样就可以在默认的 1MB 位置之外加载了。

注意：如果 CONFIG_RELOCATABLE=y， 那么 内核将会从其被加载的位置运行，编译时的物理地址 (CONFIG_PH
YSICAL_START) 将会被作为最低地址位置的限制。
```

简单来说，这意味着相同配置下的 Linux 内核可以从不同地址被启动。这是通过将程序以 位置无关代码 的形式编译来达到的。如果我们参考 /arch/x86/boot/compressed/Makefile，我们将会看到解压器的确是用 `-fPIC` 标记编译的：

```
KBUILD_CFLAGS += -fno-strict-aliasing -fPIC
```

当我们使用位置无关代码时，一段代码的地址是由一个控制地址加上程序计数器计算得到的。我们可以从任意一个地址加载使用这种方式寻址的代码。这就是为什么我们需要获得 `startup_32` 的实际地址。现在让我们回到 Linux 内核代码。我们目前的目标是计算出内核解压的地址。这个地址的计算取决于内核配置项 `CONFIG_RELOCATABLE` 。让我们看代码：

```
#ifdef CONFIG_RELOCATABLE
    movl    %ebp, %ebx
    movl    BP_kernel_alignment(%esi), %eax
    decl    %eax
    addl    %eax, %ebx
    notl    %eax
    andl    %eax, %ebx
    cmpl    $LOAD_PHYSICAL_ADDR, %ebx
    jge    1f
#endif
    movl    $LOAD_PHYSICAL_ADDR, %ebx
1:
    addl    $z_extract_offset, %ebx
```

记住 `ebp` 寄存器的值就是 `startup_32` 标签的物理地址。如果在内核配置中 `CONFIG_RELOCATABLE` 内核配置项开启，我们就把这个地址放到 `ebx` 寄存器中，对齐到 `2M` 的整数倍 ，然后和 `LOAD_PHYSICAL_ADDR` 的值比较。 `LOAD_PHYSICAL_ADDR` 宏定义在头文件 arch/x86/include/asm/boot.h 中，如下：

```
#define LOAD_PHYSICAL_ADDR ((CONFIG_PHYSICAL_START \
                + (CONFIG_PHYSICAL_ALIGN - 1)) \
                & ~(CONFIG_PHYSICAL_ALIGN - 1))
```

我们可以看到该宏只是展开成对齐的 `CONFIG_PHYSICAL_ALIGN` 值，其表示了内核加载位置的物理地址。在比较了 `LOAD_PHYSICAL_ADDR` 和 `ebx` 的值之后，我们给 `startup_32` 加上偏移来获得解压内核镜像的地址。如果 `CONFIG_RELOCATABLE` 选项在内核配置时没有开启，我们就直接将默认的地址加上 `z_extract_offset` 。

在前面的操作之后， `ebp` 包含了我们加载时的地址， `ebx` 被设为内核解压缩的目标地址。

进入长模式前的准备工作

在我们得到了重定位内核镜像的基地址之后，我们需要做切换到64位模式之前的最后准备。首先，我们需要更新全局描述符表：

```
    leal    gdt(%ebp), %eax
    movl    %eax, gdt+2(%ebp)
    lgdt    gdt(%ebp)
```

在这里我们把 `ebp` 寄存器加上 `gdt` 的偏移存到 `eax` 寄存器。接下来我们把这个地址放到 `ebp` 加上 `gdt+2` 偏移的位置上，并且用 `lgdt` 指令载入 `全局描述符表` 。为了理解这个神奇的 `gdt` 偏移量，我们需要关注 `全局描述符表` 的定义。我们可以在同一个源文件中找到其定义：

```
	.data
gdt:
	.word	gdt_end - gdt
	.long	gdt
	.word	0
	.quad	0x0000000000000000	/* NULL descriptor */
	.quad	0x00af9a000000ffff	/* __KERNEL_CS */
	.quad	0x00cf92000000ffff	/* __KERNEL_DS */
	.quad	0x0080890000000000	/* TS descriptor */
	.quad   0x0000000000000000	/* TS continued */
gdt_end:
```

我们可以看到其位于 `.data` 段，并且包含了5个描述符： `null` 、内核代码段、内核数据段和其他两个任务描述符。我们已经在上一章节载入了 `全局描述符表` ，和我们现在做的差不多，但是将描述符改为 `CS.L = 1` `CS.D = 0` 从而在 `64` 位模式下执行。我们可以看到， `gdt` 的定义从两个字节开始： `gdt_end - gdt` ，代表了 `gdt` 表的最后一个字节，或者说表的范围。接下来的4个字节包含了 `gdt` 的基地址。记住 `全局描述符表` 保存在 `48位 GDTR-全局描述符表寄存器` 中，由两个部分组成：

- 全局描述符表的大小 (16位）
- 全局描述符表的基址 (32位)

所以，我们把 `gdt` 的地址放到 `eax` 寄存器，然后存到 `.long gdt` 或者 `gdt+2` 。现在我们已经建立了 `GDTR` 寄存器的结构，并且可以用 `lgdt` 指令载入 `全局描述符表` 了。

在我们载入 `全局描述符表` 之后，我们必须启用 PAE 模式。方法是将 `cr4` 寄存器的值传入 `eax` ，将第5位置1，然后再写回 `cr4` 。

```
	movl	%cr4, %eax
	orl	$X86_CR4_PAE, %eax
	movl	%eax, %cr4
```

现在我们已经接近完成进入64位模式前的所有准备工作了。最后一步是建立页表，但是在此之前，这里有一些关于长模式的知识。

长模式

长模式是 x86_64 系列处理器的原生模式。首先让我们看一看 `x86_64` 和 `x86` 的一些区别。

`64位` 模式提供了一些新特性，比如：

- 从 `r8` 到 `r15` 8个新的通用寄存器，并且所有通用寄存器都是64位的了。
- 64位指令指针 - `RIP` ;
- 新的操作模式 - 长模式;

- 64位地址和操作数;
- RIP 相对寻址 (我们将会在接下来的章节看到一个例子).

长模式是一个传统保护模式的扩展，其由两个子模式构成：

- 64位模式
- 兼容模式

为了切换到 `64位` 模式，我们需要完成以下操作：

- 启用 PAE;
- 建立页表并且将顶级页表的地址放入 `cr3` 寄存器;
- 启用 `EFER.LME` ;
- 启用分页;

我们已经通过设置 `cr4` 控制寄存器中的 `PAE` 位启动 `PAE` 了。在下一个段落，我们就要建立页表的结构了。

初期页表初始化

现在，我们已经知道了在进入 `64位` 模式之前，我们需要先建立页表，那么就让我们看看如何建立初期的 `4G` 启动页表。

注意：我不会在这里解释虚拟内存的理论，如果你想知道更多，查看本节最后的链接

Linux 内核使用 `4级` 页表，通常我们会建立6个页表：

- 1 个 `PML4` 或称为 `4级页映射` 表，包含 1 个项；
- 1 个 `PDP` 或称为 `页目录指针` 表，包含 4 个项；
- 4 个 页目录表，一共包含 `2048` 个项；

让我们看看其实现方式。首先我们在内存中为页表清理一块缓存。每个表都是 `4096` 字节，所以我们需要 `24` KB 的空间：

```
    leal    pgtable(%ebx), %edi
    xorl    %eax, %eax
    movl    $((4096*6)/4), %ecx
    rep    stosl
```

我们把和 `ebx` 相关的 `pgtable` 的地址放到 `edi` 寄存器中，清空 `eax` 寄存器，并将 `ecx` 赋值为 `6144` 。 `rep stosl` 指令将会把 `eax` 的值写到 `edi` 指向的地址，然后给 `edi` 加 4 ， `ecx` 减 4 ，重复直到 `ecx` 小于等于 0 。所以我们才把 `6144` 赋值给 `ecx` 。

`pgtable` 定义在 arch/x86/boot/compressed/head_64.S 的最后：

```
    .section ".pgtable","a",@nobits
    .balign 4096
pgtable:
    .fill 6*4096, 1, 0
```

我们可以看到，其位于 `.pgtable` 段，大小为 `24KB` 。

在我们为 `pgtable` 分配了空间之后，我们可以开始构建顶级页表 - `PML4` ：

```
    leal    pgtable + 0(%ebx), %edi
    leal    0x1007 (%edi), %eax
    movl    %eax, 0(%edi)
```

还是在这里，我们把和 `ebx` 相关的，或者说和 `startup_32` 相关的 `pgtable` 的地址放到 `ebi` 寄存器。接下来我们把相对此地址偏移 `0x1007` 的地址放到 `eax` 寄存器中。 `0x1007` 是 `PML4` 的大小 `4096` 加上 `7` 。这里的 `7` 代表了 `PML4` 的项标记。在我们这里，这些标记是 `PRESENT+RW+USER` 。在最后我们把第一个 `PDP（页目录指针）` 项的地址写到 `PML4` 中。

在接下来的一步，我们将会在 `页目录指针（PDP）` 表（3级页表）建立 4 个带有 `PRESENT+RW+USE` 标记的 `Page Directory （2级页表）` 项：

```
    leal    pgtable + 0x1000(%ebx), %edi
    leal    0x1007(%edi), %eax
    movl    $4, %ecx
1:  movl    %eax, 0x00(%edi)
    addl    $0x00001000, %eax
    addl    $8, %edi
    decl    %ecx
    jnz    1b
```

我们把 3 级页目录指针表的基地址（从 `pgtable` 表偏移 `4096` 或者 `0x1000` ）放到 `edi` ，把第一个 2 级页目录指针表的首项的地址放到 `eax` 寄存器。把 `4` 赋值给 `ecx` 寄存器，其将会作为接下来循环的计数器，然后将第一个页目录指针项写到 `edi` 指向的地址。之后， `edi` 将会包含带有标记 `0x7` 的第一个页目录指针项的地址。接下来我们就计算后面的几个页目录指针项的地址，每个占 8 字节，把地址赋值给 `eax` ，然后回到循环开头将其写入 `edi` 所在地址。建立页表结构的最后一步就是建立 `2048` 个 `2MB` 页的页表项。

```
    leal    pgtable + 0x2000(%ebx), %edi
    movl    $0x00000183, %eax
    movl    $2048, %ecx
1:  movl    %eax, 0(%edi)
    addl    $0x00200000, %eax
    addl    $8, %edi
    decl    %ecx
    jnz    1b
```

在这里我们做的几乎和上面一样，所有的表项都带着标记 - `$0x00000183` - `PRESENT + WRITE + MBZ` 。最后我们将会拥有 `2048` 个 `2MB` 大的页，或者说：

```
>>> 2048 * 0x00200000
4294967296
```

一个 `4G` 页表。我们刚刚完成我们的初期页表结构，其映射了 `4G` 大小的内存，现在我们可以把高级页表 `PML4` 的地址放到 `cr3` 寄存器中了：

```
    leal    pgtable(%ebx), %eax
    movl    %eax, %cr3
```

这样就全部结束了。所有的准备工作都已经完成，我们可以开始看如何切换到长模式了。

切换到长模式

首先我们需要设置 MSR 中的 `EFER.LME` 标记为 `0xC0000080` ：

```
    movl    $MSR_EFER, %ecx
    rdmsr
    btsl    $_EFER_LME, %eax
    wrmsr
```

在这里我们把 `MSR_EFER` 标记（在 arch/x86/include/uapi/asm/msr-index.h 中定义）放到 `ecx` 寄存器中，然后调用 `rdmsr` 指令读取 MSR 寄存器。在 `rdmsr` 执行之后，我们将会获得 `edx:eax` 中的结果值，其取决于 `ecx` 的值。我们通过 `btsl` 指令检查 `EFER_LME` 位，并且通过 `wrmsr` 指令将 `eax` 的数据写入 `MSR` 寄存器。

下一步我们将内核段代码地址入栈（我们在 GDT 中定义了），然后将 `startup_64` 的地址导入 `eax` 。

```
    pushl    $__KERNEL_CS
    leal    startup_64(%ebp), %eax
```

在这之后我们把这个地址入栈然后通过设置 `cr0` 寄存器中的 `PG` 和 `PE` 启用分页：

```
    movl    $(X86_CR0_PG | X86_CR0_PE), %eax
    movl    %eax, %cr0
```

然后执行：

```
lret
```

指令。记住前一步我们已经将 `startup_64` 函数的地址入栈，在 `lret` 指令之后，CPU 取出了其地址跳转到那里。

这些步骤之后我们最后来到了64位模式：

```
    .code64
    .org 0x200
ENTRY(startup_64)
....
....
....
```

就是这样！

总结

这是 linux 内核启动流程的第4部分。如果你有任何的问题或者建议，你可以留言，也可以直接发消息给我 twitter 或者创建一个 issue。

下一节我们将会看到内核解压缩流程和其他更多。

如果你发现文中描述有任何问题，请提交一个 **PR** 到 **linux-insides-zh** 。

相关链接

- Protected mode
- Intel® 64 and IA-32 Architectures Software Developer's Manual 3A
- GNU linker
- SSE
- Paging
- Model specific register
- .fill instruction

* Previous part
* Paging on osdev.org
* Paging Systems
* x86 Paging Tutorial

内核引导过程. Part 5.

内核解压

这是 `内核引导过程` 系列文章的第五部分。在前一部分我们看到了切换到64位模式的过程，在这一部分我们会从这里继续。我们会看到跳进内核代码的最后步骤：内核解压前的准备、重定位和直接内核解压。所以...让我们再次深入内核源码。

内核解压前的准备

我们停在了跳转到 `64位` 入口点—— `startup_64` 的跳转之前，它在源文件 arch/x86/boot/compressed/head_64.S 里面。在之前的部分，我们已经在 `startup_32` 里面看到了到 `startup_64` 的跳转：

```
	pushl	$__KERNEL_CS
	leal	startup_64(%ebp), %eax
	...
	...
	...
	pushl	%eax
	...
	...
	...
	lret
```

由于我们加载了新的 `全局描述符表` 并且在其他模式有CPU的模式转换（在我们这里是 `64位` 模式），我们可以在 `startup_64` 的开头看到数据段的建立：

```
	.code64
	.org 0x200
ENTRY(startup_64)
	xorl	%eax, %eax
	movl	%eax, %ds
	movl	%eax, %es
	movl	%eax, %ss
	movl	%eax, %fs
	movl	%eax, %gs
```

除 `cs` 之外的段寄存器在我们进入 `长模式` 时已经重置。

下一步是计算内核编译时的位置和它被加载的位置的差：

```
#ifdef CONFIG_RELOCATABLE
    leaq    startup_32(%rip), %rbp
    movl    BP_kernel_alignment(%rsi), %eax
    decl    %eax
    addq    %rax, %rbp
    notq    %rax
    andq    %rax, %rbp
    cmpq    $LOAD_PHYSICAL_ADDR, %rbp
    jge    1f
#endif
    movq    $LOAD_PHYSICAL_ADDR, %rbp
1:
    movl    BP_init_size(%rsi), %ebx
    subl    $_end, %ebx
    addq    %rbp, %rbx
```

`rbp` 包含了解压后内核的起始地址，在这段代码执行之后 `rbx` 会包含用于解压的重定位内核代码的地址。我们已经在 `startup_32` 看到类似的代码（你可以看之前的部分计算重定位地址），但是我们需要再做这个计算，因为引导加载器可以用64位引导协议，而 `startup_32` 在这种情况下不会执行。

下一步，我们可以看到栈指针的设置和标志寄存器的重置：

```
    leaq    boot_stack_end(%rbx), %rsp

    pushq    $0
    popfq
```

如上所述，`rbx` 寄存器包含了内核解压代码的起始地址，我们把这个地址的 `boot_stack_entry` 偏移地址相加放到表示栈顶指针的 `rsp` 寄存器。在这一步之后，栈就是正确的。你可以在汇编源码文件 arch/x86/boot/compressed/head_64.S 的末尾找到 `boot_stack_end` 的定义：

```
    .bss
    .balign 4
boot_heap:
    .fill BOOT_HEAP_SIZE, 1, 0
boot_stack:
    .fill BOOT_STACK_SIZE, 1, 0
boot_stack_end:
```

它在 `.bss` 节的末尾，就在 `.pgtable` 前面。如果你查看 arch/x86/boot/compressed/vmlinux.lds.S 链接脚本，你会找到 `.bss` 和 `.pgtable` 的定义。

由于我们设置了栈，在我们计算了解压了的内核的重定位地址后，我们可以复制压缩了的内核到以上地址。在查看细节之前，我们先看这段汇编代码：

```
    pushq    %rsi
    leaq    (_bss-8)(%rip), %rsi
    leaq    (_bss-8)(%rbx), %rdi
    movq    $_bss, %rcx
    shrq    $3, %rcx
    std
    rep    movsq
    cld
    popq    %rsi
```

首先我们把 `rsi` 压进栈。我们需要保存 `rsi` 的值，因为这个寄存器现在存放指向 `boot_params` 的指针，这是包含引导相关数据的实模式结构体（你一定记得这个结构体，我们在开始设置内核的时候就填充了它）。在代码的结尾，我们会重新恢复指向 `boot_params` 的指针到 `rsi`.

接下来两个 `leaq` 指令用 `_bss - 8` 偏移和 `rip` 和 `rbx` 计算有效地址并存放到 `rsi` 和 `rdi`. 我们为什么要计算这些地址？实际上，压缩了的代码镜像存放在这份复制了的代码（从 `startup_32` 到当前的代码）和解压了的代码之间。你可以通过查看链接脚本 arch/x86/boot/compressed/vmlinux.lds.S 验证：

```
    . = 0;
    .head.text : {
        _head = . ;
        HEAD_TEXT
        _ehead = . ;
    }
    .rodata..compressed : {
        *(.rodata..compressed)
    }
    .text :    {
        _text = .;     /* Text */
        *(.text)
        *(.text.*)
        _etext = . ;
    }
```

注意 `.head.text` 节包含了 `startup_32`. 你可以从之前的部分回忆起它：

```
    __HEAD
    .code32
ENTRY(startup_32)
...
...
...
```

`.text` 节包含解压代码：

```
    .text
relocated:
...
...
...
/*
 * Do the decompression, and jump to the new kernel..
 */
...
```

`.rodata..compressed` 包含了压缩了的内核镜像。所以 `rsi` 包含 `_bss - 8` 的绝对地址，`rdi` 包含 `_bss - 8` 的重定位的相对地址。在我们把这些地址放入寄存器时，我们把 `_bss` 的地址放到了 `rcx` 寄存器。正如你在 `vmlinux.lds.S` 链接脚本中看到了一样，它和设置/内核代码一起在所有节的末尾。现在我们可以开始用 `movsq` 指令每次8字节地从 `rsi` 到 `rdi` 复制代码。

注意在数据复制前有 `std` 指令：它设置 `DF` 标志，意味着 `rsi` 和 `rdi` 会递减。换句话说，我们会从后往前复制这些字节。最后，我们用 `cld` 指令清除 `DF` 标志，并恢复 `boot_params` 到 `rsi`.

现在我们有 `.text` 节的重定位后的地址，我们可以跳到那里：

```
    leaq    relocated(%rbx), %rax
    jmp    *%rax
```

在内核解压前的最后准备

在上一段我们看到了 `.text` 节从 `relocated` 标签开始。它做的第一件事是清空 `.bss` 节：

```
    xorl    %eax, %eax
    leaq    _bss(%rip), %rdi
    leaq    _ebss(%rip), %rcx
    subq    %rdi, %rcx
    shrq    $3, %rcx
    rep    stosq
```

我们要初始化 `.bss` 节，因为我们很快要跳转到C代码。这里我们就清空 `eax`，把 `_bss` 的地址放到 `rdi`，把 `_ebss` 放到 `rcx`，然后用 `rep stosq` 填零。

最后，我们可以调用 `extract_kernel` 函数：

```
    pushq    %rsi
    movq    %rsi, %rdi
    leaq    boot_heap(%rip), %rsi
    leaq    input_data(%rip), %rdx
    movl    $z_input_len, %ecx
    movq    %rbp, %r8
    movq    $z_output_len, %r9
    call    extract_kernel
    popq    %rsi
```

我们再一次设置 `rdi` 为指向 `boot_params` 结构体的指针并把它保存到栈中。同时我们设置 `rsi` 指向用于内核解压的区域。最后一步是准备 `extract_kernel` 的参数并调用这个解压内核的函数。 `extract_kernel` 函数在 arch/x86/boot/compressed/misc.c 源文件定义并有六个参数：

- `rmode` - 指向 boot_params 结构体的指针， `boot_params` 被引导加载器填充或在早期内核初始化时填充
- `heap` - 指向早期启动堆的起始地址 `boot_heap` 的指针
- `input_data` - 指向压缩的内核，即 `arch/x86/boot/compressed/vmlinux.bin.bz2` 的指针
- `input_len` - 压缩的内核的大小
- `output` - 解压后内核的起始地址
- `output_len` - 解压后内核的大小

所有参数根据 System V Application Binary Interface 通过寄存器传递。我们已经完成了所有的准备工作，现在我们可以看内核解压的过程。

内核解压

就像我们在之前的段落中看到了那样， `extract_kernel` 函数在源文件 arch/x86/boot/compressed/misc.c 定义并有六个参数。正如我们在之前的部分看到的，这个函数从图形/控制台初始化开始。我们要再次做这件事，因为我们不知道我们是不是从实模式开始，或者是使用了引导加载器，或者引导加载器用了32位还是64位启动协议。

在最早的初始化步骤后，我们保存空闲内存的起始和末尾地址。

```
free_mem_ptr     = heap;
free_mem_end_ptr = heap + BOOT_HEAP_SIZE;
```

在这里 `heap` 是我们在 arch/x86/boot/compressed/head_64.S 得到的 `extract_kernel` 函数的第二个参数：

```
leaq    boot_heap(%rip), %rsi
```

如上所述，`boot_heap` 定义为：

```
boot_heap:
    .fill BOOT_HEAP_SIZE, 1, 0
```

在这里 `BOOT_HEAP_SIZE` 是一个展开为 `0x10000` (对 `bzip2` 内核是 `0x400000`)的宏，代表堆的大小。

在堆指针初始化后，下一步是从 arch/x86/boot/compressed/kaslr.c 调用 `choose_random_location` 函数。我们可以从函数名猜到，它选择内核镜像解压到的内存地址。看起来很奇怪，我们要寻找甚至是 `选择` 内核解压的地址，但是Linux内核支持kASLR，为了安全，它允许解压内核到随机的地址。

在这一部分，我们不会考虑Linux内核的加载地址的随机化，我们会在下一部分讨论。

现在我们回头看 misc.c. 在获得内核镜像的地址后，需要有一些检查以确保获得的随机地址是正确对齐的，并且地址没有错误：

```
if ((unsigned long)output & (MIN_KERNEL_ALIGN - 1))
    error("Destination physical address inappropriately aligned");

if (virt_addr & (MIN_KERNEL_ALIGN - 1))
    error("Destination virtual address inappropriately aligned");

if (heap > 0x3fffffffffffUL)
    error("Destination address too large");

if (virt_addr + max(output_len, kernel_total_size) > KERNEL_IMAGE_SIZE)
    error("Destination virtual address is beyond the kernel mapping area");

if ((unsigned long)output != LOAD_PHYSICAL_ADDR)
    error("Destination address does not match LOAD_PHYSICAL_ADDR");

if (virt_addr != LOAD_PHYSICAL_ADDR)
    error("Destination virtual address changed when not relocatable");
```

在所有这些检查后，我们可以看到熟悉的消息：

```
Decompressing Linux...
```

然后调用解压内核的 `__decompress` 函数：

```
__decompress(input_data, input_len, NULL, NULL, output, output_len, NULL, error);
```

`__decompress` 函数的实现取决于在内核编译期间选择什么压缩算法：

```
#ifdef CONFIG_KERNEL_GZIP
#include "../../../../lib/decompress_inflate.c"
#endif

#ifdef CONFIG_KERNEL_BZIP2
#include "../../../../lib/decompress_bunzip2.c"
#endif

#ifdef CONFIG_KERNEL_LZMA
#include "../../../../lib/decompress_unlzma.c"
#endif

#ifdef CONFIG_KERNEL_XZ
#include "../../../../lib/decompress_unxz.c"
#endif

#ifdef CONFIG_KERNEL_LZO
#include "../../../../lib/decompress_unlzo.c"
#endif

#ifdef CONFIG_KERNEL_LZ4
#include "../../../../lib/decompress_unlz4.c"
#endif
```

在内核解压之后，最后两个函数是 `parse_elf` 和 `handle_relocations` .这些函数的主要用途是把解压后的内核移动到正确的位置。事实上，解压过程会原地解压，我们还是要把内核移动到正确的地址。我们已经知道，内核镜像是一个ELF可执行文件，所以 `parse_elf` 的主要目标是移动可加载的段到正确的地址。我们可以在 `readelf` 的输出看到可加载的段：

```
readelf -l vmlinux

Elf file type is EXEC (Executable file)
Entry point 0x1000000
There are 5 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000200000 0xffffffff81000000 0x0000000001000000
                 0x0000000000893000 0x0000000000893000  R E    200000
  LOAD           0x0000000000a93000 0xffffffff81893000 0x0000000001893000
                 0x000000000016d000 0x000000000016d000  RW     200000
  LOAD           0x0000000000c00000 0x0000000000000000 0x0000000001a00000
                 0x00000000000152d8 0x00000000000152d8  RW     200000
  LOAD           0x0000000000c16000 0xffffffff81a16000 0x0000000001a16000
                 0x0000000000138000 0x000000000029b000  RWE    200000
```

`parse_elf` 函数的目标是加载这些段到从 `choose_random_location` 函数得到的 `output` 地址。这个函数从检查ELF签名标志开始：

```
Elf64_Ehdr ehdr;
Elf64_Phdr *phdrs, *phdr;

memcpy(&ehdr, output, sizeof(ehdr));

if (ehdr.e_ident[EI_MAG0] != ELFMAG0 ||
    ehdr.e_ident[EI_MAG1] != ELFMAG1 ||
    ehdr.e_ident[EI_MAG2] != ELFMAG2 ||
    ehdr.e_ident[EI_MAG3] != ELFMAG3) {
        error("Kernel is not a valid ELF file");
        return;
}
```

如果是无效的，它会打印一条错误消息并停机。如果我们得到一个有效的 `ELF` 文件，我们从给定的 `ELF` 文件遍历所有程序头，并用正确的地址复制所有可加载的段到输出缓冲区：

```
    for (i = 0; i < ehdr.e_phnum; i++) {
        phdr = &phdrs[i];

        switch (phdr->p_type) {
        case PT_LOAD:
#ifdef CONFIG_RELOCATABLE
            dest = output;
            dest += (phdr->p_paddr - LOAD_PHYSICAL_ADDR);
#else
            dest = (void *)(phdr->p_paddr);
#endif
            memmove(dest, output + phdr->p_offset, phdr->p_filesz);
            break;
        default:
            break;
        }
    }
```

这就是全部的工作。

从现在开始，所有可加载的段都在正确的位置。

在 `parse_elf` 函数之后是调用 `handle_relocations` 函数。这个函数的实现依赖于 `CONFIG_X86_NEED_RELOCS` 内核配置选项，如果它被启用，这个函数调整内核镜像的地址，只有在内核配置时启用了 `CONFIG_RANDOMIZE_BASE` 配置选项才会调用。 `handle_relocations` 函数的实现足够简单。这个函数从基准内核加载地址的值减掉 `LOAD_PHYSICAL_ADDR` 的值，从而我

们获得内核链接后要加载的地址和实际加载地址的差值。在这之后我们可以进行内核重定位，因为我们知道内核加载的实际地址、它被链接的运行的地址和内核镜像末尾的重定位表。

在内核重定位后，我们从 `extract_kernel` 回来，到 arch/x86/boot/compressed/head_64.S.

内核的地址在 `rax` 寄存器，我们跳到那里：

```
jmp    *%rax
```

就是这样。现在我们就在内核里！

结论

这是关于内核引导过程的第五部分的结尾。我们不会再看到关于内核引导的文章（可能有这篇和前面的文章的更新），但是会有关于其他内核内部细节的很多文章。

下一章会描述更高级的关于内核引导过程的细节，如加载地址随机化等等。

如果你有什么问题或建议，写个评论或在 twitter 找我。

如果你发现文中描述有任何问题，请提交一个 **PR** 到 **linux-insides-zh** 。

链接

- address space layout randomization
- initrd
- long mode
- bzip2
- RDRand instruction
- Time Stamp Counter
- Programmable Interval Timers
- Previous part

内核初始化流程

读者在这章可以了解到整个内核初始化的完整周期，从内核解压之后的第一步到内核自身运行的第一个进程。

注意 这里不是所有内核初始化步骤的介绍。这里只有通用的内核内容，不会涉及到中断控制、 ACPI 、以及其它部分。此处没有详述的部分，会在其它章节中描述。

- 内核解压之后的首要步骤 - 描述内核中的首要步骤。
- 早期的中断和异常控制 - 描述了早期的中断初始化和早期的缺页处理函数。
- 在到达内核入口之前最后的准备 - 描述了在调用 start_kernel 之前最后的准备工作。
- 内核入口 - start_kernel - 描述了内核通用代码中初始化的第一步。
- 体系架构初始化 - 描述了特定架构的初始化。
- 进一步初始化指定体系架构 - 描述了再一次的指定架构初始化流程。
- 最后对指定体系架构初始化 - 描述了指定架构初始化流程的结尾。
- 调度器初始化 - 描述了调度初始化之前的准备工作，以及调度初始化。
- RCU 初始化 - 描述了 RCU 的初始化。
- 初始化结束 - Linux内核初始化的最后部分。

内核初始化 第一部分

踏入内核代码的第一步（TODO: Need proofreading）

上一章是引导过程的最后一部分。从现在开始，我们将深入探究 Linux 内核的初始化过程。在解压缩完 Linux 内核镜像、并把它妥善地放入内存后，内核就开始工作了。我们在第一章中介绍了 Linux 内核引导程序，它的任务就是为执行内核代码做准备。而在本章中，我们将探究内核代码，看一看内核的初始化过程——即在启动 PID 为 `1` 的 `init` 进程前，内核所做的大量工作。

本章的内容很多，介绍了在内核启动前的所有准备工作。arch/x86/kernel/head_64.S 文件中定义了内核入口点，我们会从这里开始，逐步地深入下去。在 `start_kernel` 函数（定义在 init/main.c） 执行之前，我们会看到很多的初期的初始化过程，例如初期页表初始化、切换到一个新的内核空间描述符等等。

在上一章的最后一节中，我们跟踪到了 arch/x86/boot/compressed/head_64.S 文件中的 jmp 指令：

```
jmp	*%rax
```

此时 `rax` 寄存器中保存的就是 Linux 内核入口点，通过调用 `decompress_kernel` （arch/x86/boot/compressed/misc.c） 函数后获得。由此可见，内核引导程序的最后一行代码是一句指向内核入口点的跳转指令。既然已经知道了内核入口点定义在哪，我们就可以继续探究 Linux 内核在引导结束后做了些什么。

内核执行的第一步

OK，在调用了 `decompress_kernel` 函数后， `rax` 寄存器中保存了解压缩后的内核镜像的地址，并且跳转了过去。解压缩后的内核镜像的入口点定义在 arch/x86/kernel/head_64.S，这个文件的开头几行如下：

```
    __HEAD
    .code64
    .globl startup_64
startup_64:
    ...
    ...
    ...
```

我们可以看到 `startup_64` 过程定义在了 `__HEAD` 区段下。 `__HEAD` 只是一个宏，它将展开为可执行的 `.head.text` 区段：

```
#define __HEAD        .section    ".head.text","ax"
```

我们可以在 arch/x86/kernel/vmlinux.lds.S 链接器脚本文件中看到这个区段的定义：

```
.text : AT(ADDR(.text) - LOAD_OFFSET) {
    _text = .;
    ...
    ...
    ...
} :text = 0x9090
```

除了对 `.text` 区段的定义，我们还能从这个脚本文件中得知内核的默认物理地址与虚拟地址。 `_text` 是一个地址计数器，对于 x86_64 来说，它定义为：

```
. = __START_KERNEL;
```

`__START_KERNEL` 宏的定义在 arch/x86/include/asm/page_types.h 头文件中，它由内核映射的虚拟基址与基物理起始点相加得到：

```
#define __START_KERNEL    (__START_KERNEL_map + __PHYSICAL_START)

#define __PHYSICAL_START  ALIGN(CONFIG_PHYSICAL_START, CONFIG_PHYSICAL_ALIGN)
```

换句话说：

- Linux 内核的物理基址 - `0x1000000`；
- Linux 内核的虚拟基址 - `0xffffffff81000000`.

现在我们知道了 `startup_64` 过程的默认物理地址与虚拟地址，但是真正的地址必须要通过下面的代码计算得到：

```
    leaq    _text(%rip), %rbp
    subq    $_text - __START_KERNEL_map, %rbp
```

没错，虽然定义为 `0x1000000` ，但是仍然有可能变化，例如启用 kASLR 的时候。所以我们当前的目标是计算 `0x1000000` 与实际加载地址的差。这里我们首先将RIP相对地址（ `rip-relative` ）放入 `rbp` 寄存器，并且从中减去 `$_text - __START_KERNEL_map` 。我们已经知道， `_text` 在编译后的默认虚拟地址为 `0xffffffff81000000` ， 物理地址为 `0x1000000` 。 `__START_KERNEL_map` 宏将展开为 `0xffffffff80000000` ，因此对于对于第二行汇编代码，我们将得到如下的表达式：

```
rbp = 0x1000000 - (0xffffffff81000000 - 0xffffffff80000000)
```

在计算过后， `rbp` 的值将为 `0` ，代表了实际加载地址与编译后的默认地址之间的差值。在我们这个例子中， `0` 代表了 Linux 内核被加载到了默认地址，并且没有启用 kASLR 。

在得到了 `startup_64` 的地址后，我们需要检查这个地址是否已经正确对齐。下面的代码将进行这项工作：

```
    testl    $~PMD_PAGE_MASK, %ebp
    jnz    bad_address
```

在这里我们将 `rbp` 寄存器的低32位与 `PMD_PAGE_MASK` 进行比较。 `PMD_PAGE_MASK` 代表中层页目录（ `Page middle directory` ）屏蔽位（相关信息请阅读 paging 一节），它的定义如下：

```
#define PMD_PAGE_MASK           (~(PMD_PAGE_SIZE-1))

#define PMD_PAGE_SIZE           (_AC(1, UL) << PMD_SHIFT)
#define PMD_SHIFT       21
```

可以很容易得出 `PMD_PAGE_SIZE` 为 `2MB` 。在这里我们使用标准公式来检查对齐问题，如果 `text` 的地址没有对齐到 `2MB` ，则跳转到 `bad_address` 。

在此之后，我们通过检查高 `18` 位来防止这个地址过大：

```
    leaq    _text(%rip), %rax
    shrq    $MAX_PHYSMEM_BITS, %rax
    jnz    bad_address
```

这个地址必须不超过 `46` 个比特，即小于2的46次方：

```
#define MAX_PHYSMEM_BITS       46
```

OK，至此我们完成了一些初步的检查，可以继续进行后续的工作了。

修正页表基地址

在开始设置 Identity 分页之前，我们需要首先修正下面的地址：

```
    addq    %rbp, early_level4_pgt + (L4_START_KERNEL*8)(%rip)
    addq    %rbp, level3_kernel_pgt + (510*8)(%rip)
    addq    %rbp, level3_kernel_pgt + (511*8)(%rip)
    addq    %rbp, level2_fixmap_pgt + (506*8)(%rip)
```

如果 `startup_64` 的值不为默认的 `0x1000000` 的话， 则包括 `early_level4_pgt` 、 `level3_kernel_pgt` 在内的很多地址都会不正确。 `rbp` 寄存器中包含的是相对地址，因此我们把它与 `early_level4_pgt` 、 `level3_kernel_pgt` 以及 `level2_fixmap_pgt` 中特定的项相加。首先我们来看一下它们的定义：

```
NEXT_PAGE(early_level4_pgt)
    .fill    511,8,0
    .quad    level3_kernel_pgt - __START_KERNEL_map + _PAGE_TABLE

NEXT_PAGE(level3_kernel_pgt)
    .fill    L3_START_KERNEL,8,0
    .quad    level2_kernel_pgt - __START_KERNEL_map + _KERNPG_TABLE
    .quad    level2_fixmap_pgt - __START_KERNEL_map + _PAGE_TABLE

NEXT_PAGE(level2_kernel_pgt)
    PMDS(0, __PAGE_KERNEL_LARGE_EXEC,
        KERNEL_IMAGE_SIZE/PMD_SIZE)

NEXT_PAGE(level2_fixmap_pgt)
    .fill    506,8,0
    .quad    level1_fixmap_pgt - __START_KERNEL_map + _PAGE_TABLE
    .fill    5,8,0

NEXT_PAGE(level1_fixmap_pgt)
    .fill    512,8,0
```

看起来很难理解，实则不然。首先我们来看一下 `early_level4_pgt` 。它的前 (4096 - 8) 个字节全为 `0` ，即它的前 `511` 个项均不使用，之后的一项是 `level3_kernel_pgt - __START_KERNEL_map + _PAGE_TABLE` 。我们知道 `__START_KERNEL_map` 是内核的虚拟基地址，因此减去 `__START_KERNEL_map` 后就得到了 `level3_kernel_pgt` 的物理地址。现在我们来看一下 `_PAGE_TABLE` ，它是页表项的访问权限：

```
#define _PAGE_TABLE     (_PAGE_PRESENT | _PAGE_RW | _PAGE_USER | \
                         _PAGE_ACCESSED | _PAGE_DIRTY)
```

更多信息请阅读 分页 部分.

`level3_kernel_pgt` 中保存的两项用来映射内核空间，在它的前 `510` （即 `L3_START_KERNEL` ）项均为 `0` 。这里的 `L3_START_KERNEL` 保存的是在上层页目录（Page Upper Directory）中包含 `__START_KERNEL_map` 地址的那一条索引，它等于 `510` 。后面一项 `level2_kernel_pgt - __START_KERNEL_map + _KERNPG_TABLE` 中的 `level2_kernel_pgt` 比较容易理解，它是一条页表项，包含了指向中层页目录的指针，它用来映射内核空间，并且具有如下的访问权限：

```
#define _KERNPG_TABLE   (_PAGE_PRESENT | _PAGE_RW | _PAGE_ACCESSED | \
                         _PAGE_DIRTY)
```

`level2_fixmap_pgt` 是一系列虚拟地址，它们可以在内核空间中指向任意的物理地址。它们由 `level2_fixmap_pgt` 作为入口点、 `10` MB 大小的空间用来为 vsyscalls 做映射。 `level2_kernel_pgt` 则调用了 `PDMS` 宏，在 `__START_KERNEL_map` 地址处为内核的 `.text` 创建了 `512` MB 大小的空间（这 `512` MB空间的后面是模块内存空间）。

现在，在看过了这些符号的定义之后，让我们回到本节开始时介绍的那几行代码。 `rbp` 寄存器包含了实际地址与 `startup_64` 地址之差，其中 `startup_64` 的地址是在内核链接时获得的。因此我们只需要把它与各个页表项的基地址相加，就能够得到正确的地址了。在这里这些操作如下：

```
    addq    %rbp, early_level4_pgt + (L4_START_KERNEL*8)(%rip)
    addq    %rbp, level3_kernel_pgt + (510*8)(%rip)
    addq    %rbp, level3_kernel_pgt + (511*8)(%rip)
    addq    %rbp, level2_fixmap_pgt + (506*8)(%rip)
```

换句话说， `early_level4_pgt` 的最后一项就是 `level3_kernel_pgt` ， `level3_kernel_pgt` 的最后两项分别是 `level2_kernel_pgt` 和 `level2_fixmap_pgt` ， `level2_fixmap_pgt` 的第507项就是 `level1_fixmap_pgt` 页目录。

在这之后我们就得到了：

```
early_level4_pgt[511] -> level3_kernel_pgt[0]
level3_kernel_pgt[510] -> level2_kernel_pgt[0]
level3_kernel_pgt[511] -> level2_fixmap_pgt[0]
level2_kernel_pgt[0]   -> 512 MB kernel mapping
level2_fixmap_pgt[507] -> level1_fixmap_pgt
```

需要注意的是，我们并不修正 `early_level4_pgt` 以及其他页目录的基地址，我们会在构造、填充这些页目录结构的时候修正。我们修正了页表基地址后，就可以开始构造这些页目录了。

Identity Map Paging

现在我们可以进入到对初期页表进行 Identity 映射的初始化过程了。在 Identity 映射分页中，虚拟地址会被映射到地址相同的物理地址上，即 `1 : 1`。下面我们来看一下细节。首先我们找到 `_text` 与 `_early_level4_pgt` 的 RIP 相对地址，并把他们放入 `rdi` 与 `rbx` 寄存器中。

```
    leaq    _text(%rip), %rdi
    leaq    early_level4_pgt(%rip), %rbx
```

在此之后我们使用 `rax` 保存 `_text` 的地址。同时，在全局页目录表中有一条记录中存放的是 `_text` 的地址。为了得到这条索引，我们把 `_text` 的地址右移 `PGDIR_SHIFT` 位。

```
    movq    %rdi, %rax
    shrq    $PGDIR_SHIFT, %rax

    leaq    (4096 + _KERNPG_TABLE)(%rbx), %rdx
    movq    %rdx, 0(%rbx,%rax,8)
    movq    %rdx, 8(%rbx,%rax,8)
```

其中 `PGDIR_SHIFT` 为 `39`。`PGDIR_SHIFT` 表示的是在虚拟地址下的全局页目录位的屏蔽值（mask）。下面的宏定义了所有类型的页目录的屏蔽值：

```
#define PGDIR_SHIFT     39
#define PUD_SHIFT       30
#define PMD_SHIFT       21
```

此后我们就将 `level3_kernel_pgt` 的地址放进 `rdx` 中，并将它的访问权限设置为 `_KERNPG_TABLE`（见上），然后将 `level3_kernel_pgt` 填入 `early_level4_pgt` 的两项中。

然后我们给 `rdx` 寄存器加上 `4096`（即 `early_level4_pgt` 的大小），并把 `rdi` 寄存器的值（即 `_text` 的物理地址）赋值给 `rax` 寄存器。之后我们把上层页目录中的两个项写入 `level3_kernel_pgt`：

```
    addq    $4096, %rdx
    movq    %rdi, %rax
    shrq    $PUD_SHIFT, %rax
    andl    $(PTRS_PER_PUD-1), %eax
    movq    %rdx, 4096(%rbx,%rax,8)
    incl    %eax
    andl    $(PTRS_PER_PUD-1), %eax
    movq    %rdx, 4096(%rbx,%rax,8)
```

下一步我们把中层页目录表项的地址写入 `level2_kernel_pgt` ，然后修正内核的 text 和 data 的虚拟地址：

```
    leaq    level2_kernel_pgt(%rip), %rdi
    leaq    4096(%rdi), %r8
1:    testq    $1, 0(%rdi)
    jz    2f
    addq    %rbp, 0(%rdi)
2:    addq    $8, %rdi
    cmp    %r8, %rdi
    jne    1b
```

这里首先把 `level2_kernel_pgt` 的地址赋值给 `rdi` ，并把页表项的地址赋值给 `r8` 寄存器。下一步我们来检查 `level2_kernel_pgt` 中的存在位，如果其为0，就把 `rdi` 加上8以便指向下一个页。然后我们将其与 `r8` （即页表项的地址）作比较，不相等的话就跳转回前面的标签 `1` ，反之则继续运行。

接下来我们使用 `rbp` （即 `_text` 的物理地址）来修正 `phys_base` 物理地址。将 `early_level4_pgt` 的物理地址与 `rbp` 相加，然后跳转至标签 `1` ：

```
    addq    %rbp, phys_base(%rip)
    movq    $(early_level4_pgt - __START_KERNEL_map), %rax
    jmp 1f
```

其中 `phys_base` 与 `level2_kernel_pgt` 第一项相同，为 `512` MB的内核映射。

跳转至内核入口点之前的最后准备

此后我们就跳转至标签 `1` 来开启 `PAE` 和 `PGE` （Paging Global Extension），并且将 `phys_base` 的物理地址（见上）放入 `rax` 就寄存器，同时将其放入 `cr3` 寄存器：

```
1:
    movl    $(X86_CR4_PAE | X86_CR4_PGE), %ecx
    movq    %rcx, %cr4

    addq    phys_base(%rip), %rax
    movq    %rax, %cr3
```

接下来我们检查CPU是否支持 NX 位：

```
    movl    $0x80000001, %eax
    cpuid
    movl    %edx,%edi
```

首先将 `0x80000001` 放入 `eax` 中，然后执行 `cpuid` 指令来得到处理器信息。这条指令的结果会存放在 `edx` 中，我们把他再放到 `edi` 里。

现在我们把 `MSR_EFER` （即 `0xc0000080` ）放入 `ecx` ，然后执行 `rdmsr` 指令来读取CPU中的Model Specific Register (MSR)。

```
    movl    $MSR_EFER, %ecx
    rdmsr
```

返回结果将存放于 `edx:eax` 。下面展示了 `EFER` 各个位的含义：

```
63                                                                              32
 --------------------------------------------------------------------------------
|                                                                               |
|                                Reserved MBZ                                   |
|                                                                               |
 --------------------------------------------------------------------------------
31                            16  15      14      13   12  11   10  9  8 7  1   0
 --------------------------------------------------------------------------------
|                               | T |       |       |    |   |   |   |   |   |   |
| Reserved MBZ                  | C | FFXSR | LMSLE |SVME|NXE|LMA|MBZ|LME|RAZ|SCE|
|                               | E |       |       |    |   |   |   |   |   |   |
 --------------------------------------------------------------------------------
```

在这里我们不会介绍每一个位的含义，没有涉及到的位和其他的 MSR 将会在专门的部分介绍。在我们将 `EFER` 读入 `edx:eax` 之后，通过 `btsl` 来将 `_EFER_SCE` （即第0位）置1，设置 `SCE` 位将会启用 `SYSCALL` 以及 `SYSRET` 指令。下一步我们检查 `edi` （即 `cpuid` 的结果（见上）） 中的第20位。如果第 `20` 位（即 `NX` 位）置位，我们就只把 `EFER_SCE` 写入MSR。

```
    btsl    $_EFER_SCE, %eax
    btl        $20,%edi
    jnc     1f
    btsl    $_EFER_NX, %eax
    btsq    $_PAGE_BIT_NX,early_pmd_flags(%rip)
1:    wrmsr
```

如果支持 NX 那么我们就把 `_EFER_NX` 也写入MSR。在设置了 NX 后，还要对 `cr0`（control register） 中的一些位进行设置：

- `X86_CR0_PE` - 系统处于保护模式;
- `X86_CR0_MP` - 与CR0的TS标志位一同控制 WAIT/FWAIT 指令的功能；
- `X86_CR0_ET` - 386允许指定外部数学协处理器为80287或80387;
- `X86_CR0_NE` - 如果置位，则启用内置的x87浮点错误报告，否则启用PC风格的x87错误检测；
- `X86_CR0_WP` - 如果置位，则CPU在特权等级为0时无法写入只读内存页;
- `X86_CR0_AM` - 当AM位置位、EFLGS中的AC位置位、特权等级为3时，进行对齐检查;
- `X86_CR0_PG` - 启用分页.

```
#define CR0_STATE    (X86_CR0_PE | X86_CR0_MP | X86_CR0_ET | \
             X86_CR0_NE | X86_CR0_WP | X86_CR0_AM | \
             X86_CR0_PG)
movl    $CR0_STATE, %eax
movq    %rax, %cr0
```

为了从汇编执行C语言代码，我们需要建立一个栈。首先将栈指针 指向一个内存中合适的区域，然后重置FLAGS寄存器

```
movq stack_start(%rip), %rsp
pushq $0
popfq
```

在这里最有意思的地方在于 `stack_start` 。它也定义在当前的源文件中：

```
GLOBAL(stack_start)
.quad  init_thread_union+THREAD_SIZE-8
```

对于 `GLOABL` 我们应该很熟悉了。它在 arch/x86/include/asm/linkage.h 头文件中定义如下：

```
#define GLOBAL(name)    \
        .globl name;           \
        name:
```

`THREAD_SIZE` 定义在 arch/x86/include/asm/page_64_types.h，它依赖于 `KASAN_STACK_ORDER` 的值:

```
#define THREAD_SIZE_ORDER       (2 + KASAN_STACK_ORDER)
#define THREAD_SIZE  (PAGE_SIZE << THREAD_SIZE_ORDER)
```

首先来考虑当禁用了 kasan 并且 `PAGE_SIZE` 大小为4096时的情况。此时 `THREAD_SIZE` 将为 `16` KB，代表了一个线程的栈的大小。为什么是 `线程` ？我们知道每一个进程可能会有父进程和子进程。事实上，父进程和子进程使用不同的栈空间，每一个新进程都会拥有一个新的内核栈。在Linux内核中，这个栈由 `thread_info` 结构中的一个union表示：

```
union thread_union {
         struct thread_info thread_info;
         unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

例如， `init_thread_union` 定义如下：

```
union thread_union init_thread_union __init_task_data =
    { INIT_THREAD_INFO(init_task) };
```

其中 `INIT_THREAD_INFO` 接受 `task_struct` 结构类型的参数，并进行一些初始化操作：

```
#define INIT_THREAD_INFO(tsk)        \
{                                     \
    .task         = &tsk,              \
    .flags         = 0,                 \
    .cpu         = 0,                 \
    .addr_limit     = KERNEL_DS,         \
}
```

`task_struct` 结构在内核中代表了对进程的描述。因此， `thread_union` 包含了关于一个进程的低级信息，并且其位于进程栈底：

```
+-----------------------+
|                       |
|                       |
|                       |
|     Kernel stack      |
|                       |
|                       |
|                       |
|-----------------------|
|                       |
|  struct thread_info   |
|                       |
+-----------------------+
```

需要注意的是我们在栈顶保留了 `8` 个字节的空间，用来保护对下一个内存页的非法访问。

在初期启动栈设置好之后，使用 `lgdt` 指令来更新全局描述符表：

```
lgdt    early_gdt_descr(%rip)
```

其中 `early_gdt_descr` 定义如下：

```
early_gdt_descr:
    .word    GDT_ENTRIES*8-1
early_gdt_descr_base:
    .quad    INIT_PER_CPU_VAR(gdt_page)
```

需要重新加载 `全局描述附表` 的原因是，虽然目前内核工作在用户空间的低地址中，但很快内核将会在它自己的内存地址空间中运行。下面让我们来看一下 `early_gdt_descr` 的定义。全局描述符表包含了32项，用于内核代码、数据、线程局部存储段等：

```
#define GDT_ENTRIES 32
```

现在来看一下 `early_gdt_descr_base`. 首先，`gdt_page` 的定义在arch/x86/include/asm/desc.h中:

```
struct gdt_page {
    struct desc_struct gdt[GDT_ENTRIES];
} __attribute__((aligned(PAGE_SIZE)));
```

它只包含了一项 `desc_struct` 的数组 `gdt` 。 `desc_struct` 定义如下:

```
struct desc_struct {
         union {
                 struct {
                         unsigned int a;
                         unsigned int b;
                 };
                 struct {
                         u16 limit0;
                         u16 base0;
                         unsigned base1: 8, type: 4, s: 1, dpl: 2, p: 1;
                         unsigned limit: 4, avl: 1, l: 1, d: 1, g: 1, base2: 8;
                 };
         };
 } __attribute__((packed));
```

它跟 `GDT` 描述符的定义很像。同时需要注意的是，`gdt_page` 结构是 `PAGE_SIZE` (`4096`) 对齐的，即 `gdt` 将会占用一页内存。

下面我们来看一下 `INIT_PER_CPU_VAR` ，它定义在 arch/x86/include/asm/percpu.h，只是将给定的参数与 `init_per_cpu__` 连接起来：

```
#define INIT_PER_CPU_VAR(var) init_per_cpu__##var
```

所以在宏展开之后，我们会得到 `init_per_cpu__gdt_page` 。而在 linker script 中可以发现:

```
#define INIT_PER_CPU(x) init_per_cpu__##x = x + __per_cpu_load
INIT_PER_CPU(gdt_page);
```

`INIT_PER_CPU` 扩展后也将得到 `init_per_cpu__gdt_page` 并将它的值设置为相对于 `__per_cpu_load` 的偏移量。这样，我们就得到了新GDT的正确的基地址。

per-CPU变量是2.6内核中的特性。顾名思义，当我们创建一个 `per-CPU` 变量时，每个CPU都会拥有一份它自己的拷贝，在这里我们创建的是 `gdt_page` per-CPU变量。这种类型的变量有很多有点，比如由于每个CPU都只访问自己的变量而不需要锁等。因此在多处理器的情况下，每一个处理器核心都将拥有一份自己的 `GDT` 表，其中的每一项都代表了一块内存，这块内存可以由在这个核心上运行的线程访问。这里 Concepts/per-cpu 有关于 `per-CPU` 变量的更详细的介绍。

在加载好了新的全局描述附表之后，跟之前一样我们重新加载一下各个段：

```
	xorl %eax,%eax
	movl %eax,%ds
	movl %eax,%ss
	movl %eax,%es
	movl %eax,%fs
	movl %eax,%gs
```

在所有这些步骤都结束后，我们需要设置一下 `gs` 寄存器，令它指向一个特殊的栈 `irqstack` ，用于处理中断：

```
	movl	$MSR_GS_BASE,%ecx
	movl	initial_gs(%rip),%eax
	movl	initial_gs+4(%rip),%edx
	wrmsr
```

其中， `MSR_GS_BASE` 为：

```
#define MSR_GS_BASE             0xc0000101
```

我们需要把 `MSR_GS_BASE` 放入 `ecx` 寄存器，同时利用 `wrmsr` 指令向 `eax` 和 `edx` 处的地址加载数据（即指向 `initial_gs` ）。 `cs` , `fs` , `ds` 和 `ss` 段寄存器在64位模式下不用来寻址，但 `fs` 和 `gs` 可以使用。 `fs` 和 `gs` 有一个隐含的部分（与实模式下的 `cs` 段寄存器类似），这个隐含部分存储了一个描述符，其指向 Model Specific Registers。因此上面的 `0xc0000101` 是一个 `gs.base` MSR 地址。当发生系统调用 或者 中断时，入口点处并没有内核栈，因此 `MSR_GS_BASE` 将会用来存放中断栈。

接下来我们把实模式中的 bootparam 结构的地址放入 `rdi` (要记得 `rsi` 从一开始就保存了这个结构体的指针)，然后跳转到C语言代码：

```
	movq	initial_code(%rip),%rax
	pushq	$0
	pushq	$__KERNEL_CS
	pushq	%rax
	lretq
```

这里我们把 `initial_code` 放入 `rax` 中，并且向栈里分别压入一个无用的地址、 `__KERNEL_CS` 和 `initial_code` 的地址。随后的 `lreq` 指令表示从栈上弹出返回地址并跳转。 `initial_code` 同样定义在这个文件里：

```
    .balign    8
    GLOBAL(initial_code)
    .quad    x86_64_start_kernel
    ...
    ...
    ...
```

可以看到 `initial_code` 包含了 `x86_64_start_kernel` 的地址，其定义在 arch/x86/kerne/head64.c：

```
asmlinkage __visible void __init x86_64_start_kernel(char * real_mode_data) {
    ...
    ...
    ...
}
```

这个函数接受一个参数 `real_mode_data`（刚才我们把实模式下数据的地址保存到了 `rdi` 寄存器中）。

这个函数是内核中第一个执行的C语言代码！

走进 start_kernel

在我们真正到达“内核入口点”-init/main.c中的start_kernel函数之前，我们还需要最后的准备工作：

首先在 `x86_64_start_kernel` 函数中可以看到一些检查工作：

```
BUILD_BUG_ON(MODULES_VADDR < __START_KERNEL_map);
BUILD_BUG_ON(MODULES_VADDR - __START_KERNEL_map < KERNEL_IMAGE_SIZE);
BUILD_BUG_ON(MODULES_LEN + KERNEL_IMAGE_SIZE > 2*PUD_SIZE);
BUILD_BUG_ON((__START_KERNEL_map & ~PMD_MASK) != 0);
BUILD_BUG_ON((MODULES_VADDR & ~PMD_MASK) != 0);
BUILD_BUG_ON(!(MODULES_VADDR > __START_KERNEL));
BUILD_BUG_ON(!(((MODULES_END - 1) & PGDIR_MASK) == (__START_KERNEL & PGDIR_MASK)));
BUILD_BUG_ON(__fix_to_virt(__end_of_fixed_addresses) <= MODULES_END);
```

这些检查包括：模块的虚拟地址不能低于内核 text 段基地址 `__START_KERNEL_map` ，包含模块的内核 text 段的空间大小不能小于内核镜像大小等等。`BUILD_BUG_ON` 宏定义如下：

```
#define BUILD_BUG_ON(condition) ((void)sizeof(char[1 - 2*!!(condition)]))
```

我们来理解一下这些巧妙的设计是怎么工作的。首先以第一个条件 `MODULES_VADDR < __START_KERNEL_map` 为例：`!!conditions` 等价于 `condition != 0`，这代表如果 `MODULES_VADDR < __START_KERNEL_map` 为眞，则 `!!(condition)` 为1，否则为0。执行 `2*!!(condition)` 之后数值变为 `2` 或 `0`。因此，这个宏执行完后可能产生两种不同的行为：

- 编译错误。因为我们尝试取获取一个字符数组索引为负数的变量的大小。
- 没有编译错误。

就是这么简单，通过C语言中某些常量导致编译错误的技巧实现了这一设计。

接下来 start_kernel 调用了 `cr4_init_shadow` 函数，其中存储了每个CPU中 `cr4` 的Shadow Copy。上下文切换可能会修改 `cr4` 中的位，因此需要保存每个CPU中 `cr4` 的内容。在这之后将会调用 `reset_early_page_tables` 函数，它重置了所有的全局页目录项，同时向 `cr3` 中重新写入了的全局页目录表的地址：

```
for (i = 0; i < PTRS_PER_PGD-1; i++)
    early_level4_pgt[i].pgd = 0;

next_early_pgt = 0;

write_cr3(__pa_nodebug(early_level4_pgt));
```

很快我们就会设置新的页表。在这里我们遍历了所有的全局页目录项（其中 `PTRS_PER_PGD` 为 `512`），将其设置为0。之后将 `next_early_pgt` 设置为0（会在下一篇文章中介绍细节），同时把 `early_level4_pgt` 的物理地址写入 `cr3`。`__pa_nodebug` 是一个宏，将被扩展为：

```
((unsigned long)(x) - __START_KERNEL_map + phys_base)
```

此后我们清空了从 `__bss_stop` 到 `__bss_start` 的 `_bss` 段，下一步将是建立初期 `IDT（中断描述符表）` 的处理代码，内容很多，我们将会留到下一个部分再来探究。

总结

第一部分关于Linux内核的初始化过程到这里就结束了。

如果你有任何问题或建议，请在twitter上联系我 0xAX，或者通过邮件与我沟通，还可以新开issue。

下一部分我们会看到初期中断处理程序的初始化过程、内核空间的内存映射等。

相关链接

- Model Specific Register
- Paging
- Previous part - Kernel decompression
- NX
- ASLR

内核初始化 第二部分

初期中断和异常处理

在上一个 部分 我们谈到了初期中断初始化。目前我们已经处于解压缩后的Linux内核中了，还有了用于初期启动的基本的 分页 机制。我们的目标是在内核的主体代码执行前做好准备工作。

我们已经在 本章 的 第一部分 做了一些工作，在这一部分中我们会继续分析关于中断和异常处理部分的代码。

我们在上一部分谈到了下面这个循环：

```
for (i = 0; i < NUM_EXCEPTION_VECTORS; i++)
    set_intr_gate(i, early_idt_handler_array[i]);
```

这段代码位于 arch/x86/kernel/head64.c。在分析这段代码之前，我们先来了解一些关于中断和中断处理程序的知识。

理论

中断是一种由软件或硬件产生的、向CPU发出的事件。例如，如果用户按下了键盘上的一个按键时，就会产生中断。此时CPU将会暂停当前的任务，并且将控制流转到特殊的程序中 —— 中断处理程序(Interrupt Handler)。一个中断处理程序会对中断进行处理，然后将控制权交还给之前暂停的任务中。中断分为三类：

- 软件中断 - 当一个软件可以向CPU发出信号，表明它需要系统内核的相关功能时产生。这些中断通常用于系统调用；
- 硬件中断 - 当一个硬件有任何事件发生时产生，例如键盘的按键被按下；
- 异常 - 当CPU检测到错误时产生，例如发生了除零错误或者访问了一个不存在的内存页。

每一个中断和异常都可以由一个数来表示，这个数叫做 `向量号` ，它可以取从 `0` 到 `255` 中的任何一个数。通常在实践中前 `32` 个向量号用来表示异常， `32` 到 `255` 用来表示用户定义的中断。可以看到在上面的代码中， `NUM_EXCEPTION_VECTORS` 就定义为：

```
#define NUM_EXCEPTION_VECTORS 32
```

CPU会从 APIC 或者 CPU 引脚接收中断，并使用中断向量号作为 `中断描述符表` 的索引。下面的表中列出了 `0-31` 号异常：

```
----------------------------------------------------------------------------------------
--------
|Vector|Mnemonic|Description         |Type |Error Code|Source                   |
----------------------------------------------------------------------------------------
--------
|0     | #DE    |Divide Error        |Fault|NO        |DIV and IDIV
       |
|---------------------------------------------------------------------------------------
--------
|1     | #DB    |Reserved            |F/T  |NO        |
       |
|---------------------------------------------------------------------------------------
--------
|2     | ---    |NMI                 |INT  |NO        |external NMI
       |
|---------------------------------------------------------------------------------------
--------
|3     | #BP    |Breakpoint          |Trap |NO        |INT 3
       |
|---------------------------------------------------------------------------------------
--------
|4     | #OF    |Overflow            |Trap |NO        |INTO  instruction
       |
|---------------------------------------------------------------------------------------
--------
|5     | #BR    |Bound Range Exceeded|Fault|NO        |BOUND instruction
       |
|---------------------------------------------------------------------------------------
--------
|6     | #UD    |Invalid Opcode      |Fault|NO        |UD2 instruction
       |
|---------------------------------------------------------------------------------------
--------
|7     | #NM    |Device Not Available|Fault|NO        |Floating point or [F]WAIT
       |
|---------------------------------------------------------------------------------------
--------
|8     | #DF    |Double Fault        |Abort|YES       |Ant instrctions which can gener
ate NMI|
|---------------------------------------------------------------------------------------
--------
|9     | ---    |Reserved            |Fault|NO        |
       |
|---------------------------------------------------------------------------------------
--------
|10    | #TS    |Invalid TSS         |Fault|YES       |Task switch or TSS access
       |
|---------------------------------------------------------------------------------------
--------
```

```
|11     | #NP    |Segment Not Present |Fault|NO        |Accessing segment register
        |
|-----------------------------------------------------------------------------------------
--------
|12     | #SS    |Stack-Segment Fault |Fault|YES       |Stack operations
        |
|-----------------------------------------------------------------------------------------
--------
|13     | #GP    |General Protection  |Fault|YES       |Memory reference
        |
|-----------------------------------------------------------------------------------------
--------
|14     | #PF    |Page fault          |Fault|YES       |Memory reference
        |
|-----------------------------------------------------------------------------------------
--------
|15     | ---    |Reserved            |     |NO        |
        |
|-----------------------------------------------------------------------------------------
--------
|16     | #MF    |x87 FPU fp error    |Fault|NO        |Floating point or [F]Wait
        |
|-----------------------------------------------------------------------------------------
--------
|17     | #AC    |Alignment Check     |Fault|YES       |Data reference
        |
|-----------------------------------------------------------------------------------------
--------
|18     | #MC    |Machine Check       |Abort|NO        |
        |
|-----------------------------------------------------------------------------------------
--------
|19     | #XM    |SIMD fp exception   |Fault|NO        |SSE[2,3] instructions
        |
|-----------------------------------------------------------------------------------------
--------
|20     | #VE    |Virtualization exc. |Fault|NO        |EPT violations
        |
|-----------------------------------------------------------------------------------------
--------
|21-31  | ---    |Reserved            |INT  |NO        |External interrupts
        |
------------------------------------------------------------------------------------------
--------
```

为了能够对中断进行处理，CPU使用了一种特殊的结构 - 中断描述符表（IDT）。IDT 是一个由描述符组成的数组，其中每个描述符都为8个字节，与全局描述附表一致；不过不同的是，我们把IDT中的每一项叫做 `门(gate)` 。为了获得某一项描述符的起始地址，CPU 会把向量号

乘以8，在64位模式中则会乘以16。在前面我们已经见过，CPU使用一个特殊的 `GDTR` 寄存器来存放全局描述符表的地址，中断描述符表也有一个类似的寄存器 `IDTR` ，同时还有用于将基地址加载入这个寄存器的指令 `lidt` 。

64位模式下 IDT 的每一项的结构如下：

```
127                                                                             96
 --------------------------------------------------------------------------------
|                                                                               |
|                                Reserved                                       |
|                                                                               |
 --------------------------------------------------------------------------------
95                                                                              64
 --------------------------------------------------------------------------------
|                                                                               |
|                               Offset 63..32                                   |
|                                                                               |
 --------------------------------------------------------------------------------
63                             48 47       46  44   42    39             34    32
 --------------------------------------------------------------------------------
|                                  |       |  D  |   |      |      |   |   |     |
|       Offset 31..16              |   P   |  P  | 0 |Type |0 0 0 | 0 | 0 | IST |
|                                  |       |  L  |   |      |      |   |   |     |
 --------------------------------------------------------------------------------
31                                    15 16                                      0
 --------------------------------------------------------------------------------
|                                      |                                        |
|          Segment Selector            |                 Offset 15..0           |
|                                      |                                        |
 --------------------------------------------------------------------------------
```

其中:

- `Offset` - 代表了到中断处理程序入口点的偏移；
- `DPL` - 描述符特权级别；
- `P` - Segment Present 标志;
- `Segment selector` - 在GDT或LDT中的代码段选择子；
- `IST` - 用来为中断处理提供一个新的栈。

最后的 `Type` 域描述了这一项的类型，中断处理程序共分为三种：

- 任务描述符
- 中断描述符
- 陷阱描述符

中断和陷阱描述符包含了一个指向中断处理程序的远 (far) 指针，二者唯一的不同在于CPU处理 `IF` 标志的方式。如果是由中断门进入中断处理程序的，CPU 会清除 `IF` 标志位，这样当当前中断处理程序执行时，CPU 不会对其他的中断进行处理；只有当当前的中断处理程序

返回时，CPU 才在 `iret` 指令执行时重新设置 `IF` 标志位。

中断门的其他位为保留位，必须为0。下面我们来看一下 CPU 是如何处理中断的：

- CPU 会在栈上保存标志寄存器、`cs` 段寄存器和程序计数器IP；
- 如果中断是由错误码引起的（比如 `#PF` ），CPU会在栈上保存错误码；
- 在中断处理程序执行完毕后，由 `iret` 指令返回。

OK，接下来我们继续分析代码。

设置并加载 IDT

我们分析到了如下代码：

```
for (i = 0; i < NUM_EXCEPTION_VECTORS; i++)
    set_intr_gate(i, early_idt_handler_array[i]);
```

这里循环内部调用了 `set_intr_gate` ，它接受两个参数：

- 中断号，即 `向量号` ；
- 中断处理程序的地址。

同时，这个函数还会将中断门插入至 `IDT` 表中，代码中的 `&idt_descr` 数组即为 `IDT` 。首先让我们来看一下 `early_idt_handler_array` 数组，它定义在 arch/x86/include/asm/segment.h 头文件中，包含了前32个异常处理程序的地址：

```
#define EARLY_IDT_HANDLER_SIZE   9
#define NUM_EXCEPTION_VECTORS    32

extern const char early_idt_handler_array[NUM_EXCEPTION_VECTORS][EARLY_IDT_HANDLER_SIZ
E];
```

`early_idt_handler_array` 是一个大小为 `288` 字节的数组，每一项为 `9` 个字节，其中2个字节的备用指令用于向栈中压入默认错误码（如果异常本身没有提供错误码的话），2个字节的指令用于向栈中压入向量号，剩余5个字节用于跳转到异常处理程序。

在上面的代码中，我们只通过一个循环向 `IDT` 中填入了前32项内容，这是因为在整个初期设置阶段，中断是禁用的。 `early_idt_handler_array` 数组中的每一项指向的都是同一个通用中断处理程序，定义在 arch/x86/kernel/head_64.S 。我们先暂时跳过这个数组的内容，看一下 `set_intr_gate` 的定义。

`set_intr_gate` 宏定义在 arch/x86/include/asm/desc.h：

```
#define set_intr_gate(n, addr)                          \
        do {                                                    \
                BUG_ON((unsigned)n > 0xFF);                     \
                _set_gate(n, GATE_INTERRUPT, (void *)addr, 0, 0,        \
                          __KERNEL_CS);                         \
                _trace_set_gate(n, GATE_INTERRUPT, (void *)trace_##addr,\
                                0, 0, __KERNEL_CS);             \
        } while (0)
```

首先 `BUG_ON` 宏确保了传入的中断向量号不会大于255，因为我们最多只有 `256` 个中断。然后它调用了 `_set_gate` 函数，它会将中断门写入 `IDT` ：

```
static inline void _set_gate(int gate, unsigned type, void *addr,
                             unsigned dpl, unsigned ist, unsigned seg)
{
        gate_desc s;
        pack_gate(&s, type, (unsigned long)addr, dpl, ist, seg);
        write_idt_entry(idt_table, gate, &s);
        write_trace_idt_entry(gate, &s);
}
```

在 `_set_gate` 函数的开始，它调用了 `pack_gate` 函数。这个函数会使用给定的参数填充 `gate_desc` 结构：

```
static inline void pack_gate(gate_desc *gate, unsigned type, unsigned long func,
                             unsigned dpl, unsigned ist, unsigned seg)
{
        gate->offset_low        = PTR_LOW(func);
        gate->segment           = __KERNEL_CS;
        gate->ist               = ist;
        gate->p                 = 1;
        gate->dpl               = dpl;
        gate->zero0             = 0;
        gate->zero1             = 0;
        gate->type              = type;
        gate->offset_middle     = PTR_MIDDLE(func);
        gate->offset_high       = PTR_HIGH(func);
}
```

在这个函数里，我们把从主循环中得到的中断处理程序入口点地址拆成三个部分，填入门描述符中。下面的三个宏就用来做这个拆分工作：

```
#define PTR_LOW(x) ((unsigned long long)(x) & 0xFFFF)
#define PTR_MIDDLE(x) (((unsigned long long)(x) >> 16) & 0xFFFF)
#define PTR_HIGH(x) ((unsigned long long)(x) >> 32)
```

调用 `PTR_LOW` 可以得到 x 的低 `2` 个字节，调用 `PTR_MIDDLE` 可以得到 x 的中间 `2` 个字节，调用 `PTR_HIGH` 则能够得到 x 的高 `4` 个字节。接下来我们来位中断处理程序设置段选择子，即内核代码段 `__KERNEL_CS` 。然后将 `Interrupt Stack Table` 和 `描述符特权等级` （最高特权等级）设置为0，以及在最后设置 `GAT_INTERRUPT` 类型。

现在我们已经设置好了IDT中的一项，那么通过调用 `native_write_idt_entry` 函数来把复制到 `IDT` ：

```
static inline void native_write_idt_entry(gate_desc *idt, int entry, const gate_desc *
gate)
{
        memcpy(&idt[entry], gate, sizeof(*gate));
}
```

主循环结束后， `idt_table` 就已经设置完毕了，其为一个 `gate_desc` 数组。然后我们就可以通过下面的代码加载 `中断描述符表` ：

```
load_idt((const struct desc_ptr *)&idt_descr);
```

其中， `idt_descr` 为：

```
struct desc_ptr idt_descr = { NR_VECTORS * 16 - 1, (unsigned long) idt_table };
```

`load_idt` 函数只是执行了一下 `lidt` 指令：

```
asm volatile("lidt %0"::"m" (*dtr));
```

你可能已经注意到了，在代码中还有对 `_trace_*` 函数的调用。这些函数会用跟 `_set_gate` 同样的方法对 `IDT` 门进行设置，但仅有一处不同：这些函数并不设置 `idt_table` ，而是 `trace_idt_table` ，用于设置追踪点（tracepoint，我们将会在其他章节介绍这一部分）。

好了，至此我们已经了解到，通过设置并加载 `中断描述符表` ，能够让CPU在发生中断时做出相应的动作。下面让我们来看一下如何编写中断处理程序。

初期中断处理程序

在上面的代码中，我们用 `early_idt_handler_array` 的地址来填充了 `IDT` ，这个 `early_idt_handler_array` 定义在 arch/x86/kernel/head_64.S：

```
    .globl early_idt_handler_array
early_idt_handlers:
    i = 0
    .rept NUM_EXCEPTION_VECTORS
    .if (EXCEPTION_ERRCODE_MASK >> i) & 1
    pushq $0
    .endif
    pushq $i
    jmp early_idt_handler_common
    i = i + 1
    .fill early_idt_handler_array + i*EARLY_IDT_HANDLER_SIZE - ., 1, 0xcc
    .endr
```

这段代码自动生成为前 `32` 个异常生成了中断处理程序。首先，为了统一栈的布局，如果一个异常没有返回错误码，那么我们就手动在栈中压入一个 `0` 。然后再在栈中压入中断向量号，最后跳转至通用的中断处理程序 `early_idt_handler_common` 。我们可以通过 `objdump` 命令的输出一探究竟：

```
$ objdump -D vmlinux
...
...
...
ffffffff81fe5000 <early_idt_handler_array>:
ffffffff81fe5000:       6a 00                   pushq  $0x0
ffffffff81fe5002:       6a 00                   pushq  $0x0
ffffffff81fe5004:       e9 17 01 00 00          jmpq   ffffffff81fe5120 <early_idt_han
dler_common>
ffffffff81fe5009:       6a 00                   pushq  $0x0
ffffffff81fe500b:       6a 01                   pushq  $0x1
ffffffff81fe500d:       e9 0e 01 00 00          jmpq   ffffffff81fe5120 <early_idt_han
dler_common>
ffffffff81fe5012:       6a 00                   pushq  $0x0
ffffffff81fe5014:       6a 02                   pushq  $0x2
...
...
...
```

由于在中断发生时，CPU 会在栈上压入标志寄存器、 `CS` 段寄存器和 `RIP` 寄存器的内容。因此在 `early_idt_handler` 执行前，栈的布局如下：

```
|--------------------|
| %rflags            |
| %cs                |
| %rip               |
| rsp --> error code |
|--------------------|
```

下面我们来看一下 `early_idt_handler_common` 的实现。它也定义在 arch/x86/kernel/head_64.S 文件中。首先它会检查当前中断是否为 不可屏蔽中断(NMI)，如果是则简单地忽略它们：

```
	cmpl $2,(%rsp)
	je .Lis_nmi
```

其中 `is_nmi` 为:

```
is_nmi:
	addq $16,%rsp
	INTERRUPT_RETURN
```

这段程序首先从栈顶弹出错误码和中断向量号，然后通过调用 `INTERRUPT_RETURN` ，即 `iretq` 指令直接返回。

如果当前中断不是 `NMI` ，则首先检查 `early_recursion_flag` 以避免在 `early_idt_handler_common` 程序中递归地产生中断。如果一切都没问题，就先在栈上保存通用寄存器，为了防止中断返回时寄存器的内容错乱：

```
	pushq %rax
	pushq %rcx
	pushq %rdx
	pushq %rsi
	pushq %rdi
	pushq %r8
	pushq %r9
	pushq %r10
	pushq %r11
```

然后我们检查栈上的段选择子：

```
	cmpl $__KERNEL_CS,96(%rsp)
	jne 11f
```

段选择子必须为内核代码段，如果不是则跳转到标签 `11` ，输出 `PANIC` 信息并打印栈的内容。然后我们来检查向量号，如果是 `#PF` 即 缺页中断（Page Fault），那么就把 `cr2` 寄存器中的值赋值给 `rdi` ，然后调用 `early_make_pgtable` （详见后文）：

```
    cmpl $14,72(%rsp)
    jnz 10f
    GET_CR2_INTO(%rdi)
    call early_make_pgtable
    andl %eax,%eax
    jz 20f
```

如果向量号不是 `#PF` ，那么就恢复通用寄存器：

```
    popq %r11
    popq %r10
    popq %r9
    popq %r8
    popq %rdi
    popq %rsi
    popq %rdx
    popq %rcx
    popq %rax
```

并调用 `iret` 从中断处理程序返回。

第一个中断处理程序到这里就结束了。由于它只是一个初期中段处理程序，因此只处理缺页中断。下面让我们首先来看一下缺页中断处理程序，其他中断的处理程序我们之后再进行分析。

缺页中断处理程序

在上一节中我们第一次见到了初期中断处理程序，它检查了缺页中断的中断号，并调用了 `early_make_pgtable` 来建立新的页表。在这里我们需要提供 `#PF` 中断处理程序，以便为之后将内核加载至 `4G` 地址以上，并且能访问位于4G以上的 `boot_params` 结构体。

`early_make_pgtable` 的实现在 arch/x86/kernel/head64.c，它接受一个参数：从 `cr2` 寄存器得到的地址，这个地址引发了内存中断。下面让我们来看一下：

```
int __init early_make_pgtable(unsigned long address)
{
    unsigned long physaddr = address - __PAGE_OFFSET;
    unsigned long i;
    pgdval_t pgd, *pgd_p;
    pudval_t pud, *pud_p;
    pmdval_t pmd, *pmd_p;
    ...
    ...
    ...
}
```

首先它定义了一些 `*val_t` 类型的变量。这些类型均为：

```
typedef unsigned long   pgdval_t;
```

此外，我们还会遇见 `*_t` (不带val)的类型，比如 `pgd_t` ……这些类型都定义在 arch/x86/include/asm/pgtable_types.h，形式如下：

```
typedef struct { pgdval_t pgd; } pgd_t;
```

例如，

```
extern pgd_t early_level4_pgt[PTRS_PER_PGD];
```

在这里 `early_level4_pgt` 代表了初期顶层页表目录，它是一个 `pdg_t` 类型的数组，其中的 `pgd` 指向了下一级页表。

在确认不是非法地址后，我们取得页表中包含引起 `#PF` 中断的地址的那一项，将其赋值给 `pgd` 变量：

```
pgd_p = &early_level4_pgt[pgd_index(address)].pgd;
pgd = *pgd_p;
```

接下来我们检查一下 `pgd` ，如果它包含了正确的全局页表项的话，我们就把这一项的物理地址处理后赋值给 `pud_p` ：

```
pud_p = (pudval_t *)((pgd & PTE_PFN_MASK) + __START_KERNEL_map - phys_base);
```

其中 `PTE_PFN_MASK` 是一个宏：

```
#define PTE_PFN_MASK            ((pteval_t)PHYSICAL_PAGE_MASK)
```

展开后将为：

```
(~(PAGE_SIZE-1)) & ((1 << 46) - 1)
```

或者写为：

```
0b1111111111111111111111111111111111111111111111
```

它是一个46bit大小的页帧屏蔽值。

如果 `pgd` 没有包含有效的地址，我们就检查 `next_early_pgt` 与 `EARLY_DYNAMIC_PAGE_TABLES` （即 `64` ）的大小。 `EARLY_DYNAMIC_PAGE_TABLES` 它是一个固定大小的缓冲区，用来在需要的时候建立新的页表。如果 `next_early_pgt` 比 `EARLY_DYNAMIC_PAGE_TABLES` 大，我们就用一个上层页目录指针指向当前的动态页表，并将它的物理地址与 `_KERPG_TABLE` 访问权限一起写入全局页目录表：

```
if (next_early_pgt >= EARLY_DYNAMIC_PAGE_TABLES) {
    reset_early_page_tables();
    goto again;
}

pud_p = (pudval_t *)early_dynamic_pgts[next_early_pgt++];
for (i = 0; i < PTRS_PER_PUD; i++)
    pud_p[i] = 0;
*pgd_p = (pgdval_t)pud_p - __START_KERNEL_map + phys_base + _KERNPG_TABLE;
```

然后我们来修正上层页目录的地址：

```
pud_p += pud_index(address);
pud = *pud_p;
```

下面我们对中层页目录重复上面同样的操作。最后我们利用 In the end we fix address of the page middle directory which contains maps kernel text+data virtual addresses:

```
pmd = (physaddr & PMD_MASK) + early_pmd_flags;
pmd_p[pmd_index(address)] = pmd;
```

到此缺页中断处理程序就完成了它所有的工作，此时 `early_level4_pgt` 就包含了指向合法地址的项。

小结

本书的第二部分到此结束了。

如果你有任何问题或建议，请在twitter上联系我 0xAX，或者通过邮件与我沟通，还可以新开issue。

接下来我们将会看到进入内核入口点 `start_kernel` 函数之前剩下所有的准备工作。

相关链接

- GNU assembly .rept

* APIC
* NMI
* Page table
* Interrupt handler
* Page Fault,
* Previous part

内核初始化 第三部分

进入内核入口点之前最后的准备工作

这是 Linux 内核初始化过程的第三部分。在上一个部分 中我们接触到了初期中断和异常处理，而在这个部分中我们要继续看一看 Linux 内核的初始化过程。在之后的章节我们将会关注“内核入口点”—— init/main.c 文件中的 `start_kernel` 函数。没错，从技术上说这并不是内核的入口点，只是不依赖于特定架构的通用内核代码的开始。不过，在我们调用 `start_kernel` 之前，有些准备必须要做。下面我们就来看一看。

boot_params again

在上一个部分中我们讲到了设置中断描述符表，并将其加载进 `IDTR` 寄存器。下一步是调用 `copy_bootdata` 函数：

```
copy_bootdata(__va(real_mode_data));
```

这个函数接受一个参数—— `read_mode_data` 的虚拟地址。 `boot_params` 结构体是在 arch/x86/include/uapi/asm/bootparam.h 作为第一个参数传递到 arch/x86/kernel/head_64.S 中的 `x86_64_start_kernel` 函数的：

```
	/* rsi is pointer to real mode structure with interesting info.
	   pass it to C */
	movq	%rsi, %rdi
```

下面我们来看一看 `__va` 宏。 这个宏定义在 init/main.c：

```
#define __va(x)                 ((void *)((unsigned long)(x)+PAGE_OFFSET))
```

其中 `PAGE_OFFSET` 就是 `__PAGE_OFFSET` （即 `0xffff880000000000` ），也是所有对物理地址进行直接映射后的虚拟基地址。因此我们就得到了 `boot_params` 结构体的虚拟地址，并把他传入 `copy_bootdata` 函数中。在这个函数里我们把 `real_mod_data` （定义在 arch/x86/kernel/setup.h） 拷贝进 `boot_params` ：

```
extern struct boot_params boot_params;
```

`copy_boot_data` 的实现如下:

```
static void __init copy_bootdata(char *real_mode_data)
{
    char * command_line;
    unsigned long cmd_line_ptr;

    memcpy(&boot_params, real_mode_data, sizeof boot_params);
    sanitize_boot_params(&boot_params);
    cmd_line_ptr = get_cmd_line_ptr();
    if (cmd_line_ptr) {
        command_line = __va(cmd_line_ptr);
        memcpy(boot_command_line, command_line, COMMAND_LINE_SIZE);
    }
}
```

首先，这个函数的声明中有一个 `__init` 前缀，这表示这个函数只在初始化阶段使用，并且它所使用的内存将会被释放。

在这个函数中首先声明了两个用于解析内核命令行的变量，然后使用 `memcpy` 函数将 `real_mode_data` 拷贝进 `boot_params` 。如果系统引导工具（bootloader）没能正确初始化 `boot_params` 中的某些成员的话，那么在接下来调用的 `sanitize_boot_params` 函数中将会对这些成员进行清零，比如 `ext_ramdisk_image` 等。此后我们通过调用 `get_cmd_line_ptr` 函数来得到命令行的地址：

```
unsigned long cmd_line_ptr = boot_params.hdr.cmd_line_ptr;
cmd_line_ptr |= (u64)boot_params.ext_cmd_line_ptr << 32;
return cmd_line_ptr;
```

`get_cmd_line_ptr` 函数将会从 `boot_params` 中获得命令行的64位地址并返回。最后，我们检查一下是否正确获得了 `cmd_line_ptr` ，并把它的虚拟地址拷贝到一个字节数组 `boot_command_line` 中：

```
extern char __initdata boot_command_line[];
```

这一步完成之后，我们就得到了内核命令行和 `boot_params` 结构体。之后，内核通过调用 `load_ucode_bsp` 函数来加载处理器微代码（microcode），不过我们目前先暂时忽略这一步。

微代码加载之后，内核会对 `console_loglevel` 进行检查，同时通过 `early_printk` 函数来打印出字符串 `Kernel Alive` 。不过这个输出不会真的被显示出来，因为这个时候 `early_printk` 还没有被初始化。这是目前内核中的一个小bug，作者已经提交了补丁 commit，补丁很快就能应用在主分支中了。所以你可以先跳过这段代码。

初始化内存页

至此，我们已经拷贝了 `boot_params` 结构体，接下来将对初期页表进行一些设置以便在初始化内核的过程中使用。我们之前已经对初始化了初期页表，以便支持换页，这在之前的部分中已经讨论过。现在则通过调用 `reset_early_page_tables` 函数将初期页表中大部分项清零（在之前的部分也有介绍），只保留内核高地址的映射。然后我们调用：

```
clear_page(init_level4_pgt);
```

`init_level4_pgt` 同样定义在 arch/x86/kernel/head_64.S:

```
NEXT_PAGE(init_level4_pgt)
    .quad   level3_ident_pgt - __START_KERNEL_map + _KERNPG_TABLE
    .org    init_level4_pgt + L4_PAGE_OFFSET*8, 0
    .quad   level3_ident_pgt - __START_KERNEL_map + _KERNPG_TABLE
    .org    init_level4_pgt + L4_START_KERNEL*8, 0
    .quad   level3_kernel_pgt - __START_KERNEL_map + _PAGE_TABLE
```

这段代码为内核的代码段、数据段和 bss 段映射了前 2.5G 个字节。`clear_page` 函数定义在 arch/x86/lib/clear_page_64.S：

```
ENTRY(clear_page)
    CFI_STARTPROC
    xorl %eax,%eax
    movl $4096/64,%ecx
    .p2align 4
    .Lloop:
    decl    %ecx
#define PUT(x) movq %rax,x*8(%rdi)
    movq %rax,(%rdi)
    PUT(1)
    PUT(2)
    PUT(3)
    PUT(4)
    PUT(5)
    PUT(6)
    PUT(7)
    leaq 64(%rdi),%rdi
    jnz    .Lloop
    nop
    ret
    CFI_ENDPROC
    .Lclear_page_end:
    ENDPROC(clear_page)
```

顾名思义，这个函数会将页表清零。这个函数的开始和结束部分有两个宏 `CFI_STARTPROC` 和 `CFI_ENDPROC`，他们会展开成 GNU 汇编指令，用于调试：

```
#define CFI_STARTPROC           .cfi_startproc
#define CFI_ENDPROC             .cfi_endproc
```

在 `CFI_STARTPROC` 之后我们将 `eax` 寄存器清零，并将 `ecx` 赋值为 64（用作计数器）。接下来从 `.Lloop` 标签开始循环，首先就是将 `ecx` 减一。然后将 `rax` 中的值（目前为0）写入 `rdi` 指向的地址，`rdi` 中保存的是 `init_level4_pgt` 的基地址。接下来重复7次这个步骤，但是每次都相对 `rdi` 多偏移8个字节。之后 `init_level4_pgt` 的前64个字节就都被填充为0了。接下来我们将 `rdi` 中的值加上64，重复这个步骤，直到 `ecx` 减至0。最后就完成了将 `init_level4_pgt` 填零。

在将 `init_level4_pgt` 填0之后，再把它的最后一项设置为内核高地址的映射：

```
init_level4_pgt[511] = early_level4_pgt[511];
```

在前面我们已经使用 `reset_early_page_table` 函数清除 `early_level4_pgt` 中的大部分项，而只保留内核高地址的映射。

`x86_64_start_kernel` 函数的最后一步是调用：

```
x86_64_start_reservations(real_mode_data);
```

并传入 `real_mode_data` 参数。`x86_64_start_reservations` 函数与 `x86_64_start_kernel` 函数定义在同一个文件中：

```
void __init x86_64_start_reservations(char *real_mode_data)
{
    if (!boot_params.hdr.version)
        copy_bootdata(__va(real_mode_data));

    reserve_ebda_region();

    start_kernel();
}
```

这就是进入内核入口点之前的最后一个函数了。下面我们就来介绍一下这个函数。

内核入口点前的最后一步

在 `x86_64_start_reservations` 函数中首先检查了 `boot_params.hdr.version`：

```
if (!boot_params.hdr.version)
    copy_bootdata(__va(real_mode_data));
```

如果它为0，则再次调用 `copy_bootdata` ，并传入 `real_mode_data` 的虚拟地址。

接下来则调用了 `reserve_ebda_region` 函数，它定义在 arch/x86/kernel/head.c。这个函数为 `EBDA` （即Extended BIOS Data Area，扩展BIOS数据区域）预留空间。扩展BIOS预留区域位于常规内存顶部（译注：常规内存（Conventiional Memory）是指前640K字节内存），包含了端口、磁盘参数等数据。

接下来我们来看一下 `reserve_ebda_region` 函数。它首先会检查是否启用了半虚拟化：

```
if (paravirt_enabled())
    return;
```

如果开启了半虚拟化，那么就退出 `reserve_ebda_region` 函数，因为此时没有扩展BIOS数据区域。下面我们首先得到低地址内存的末尾地址：

```
lowmem = *(unsigned short *)__va(BIOS_LOWMEM_KILOBYTES);
lowmem <<= 10;
```

首先我们得到了BIOS地地址内存的虚拟地址，以KB为单位，然后将其左移10位（即乘以1024）转换为以字节为单位。然后我们需要获得扩展BIOS数据区域的地址：

```
ebda_addr = get_bios_ebda();
```

其中， `get_bios_ebda` 函数定义在 arch/x86/include/asm/bios_ebda.h：

```
static inline unsigned int get_bios_ebda(void)
{
    unsigned int address = *(unsigned short *)phys_to_virt(0x40E);
    address <<= 4;
    return address;
}
```

下面我们来尝试理解一下这段代码。这段代码中，首先我们将物理地址 `0x40E` 转换为虚拟地址， `0x0040:0x000e` 就是包含有扩展BIOS数据区域基地址的代码段。这里我们使用了 `phys_to_virt` 函数进行地址转换，而不是之前使用的 `__va` 宏。不过，事实上他们两个基本上是一样的：

```
static inline void *phys_to_virt(phys_addr_t address)
{
        return __va(address);
}
```

而不同之处在于，`phys_to_virt` 函数的参数类型 `phys_addr_t` 的定义依赖于 `CONFIG_PHYS_ADDR_T_64BIT`：

```
#ifdef CONFIG_PHYS_ADDR_T_64BIT
    typedef u64 phys_addr_t;
#else
    typedef u32 phys_addr_t;
#endif
```

具体的类型是由 `CONFIG_PHYS_ADDR_T_64BIT` 设置选项控制的。此后我们得到了包含扩展BIOS数据区域虚拟基地址的段，把它左移4位后返回。这样，`ebda_addr` 变量就包含了扩展BIOS数据区域的基地址。

下一步我们来检查扩展BIOS数据区域与低地址内存的地址，看一看它们是否小于 `INSANE_CUTOFF` 宏：

```
if (ebda_addr < INSANE_CUTOFF)
    ebda_addr = LOWMEM_CAP;

if (lowmem < INSANE_CUTOFF)
    lowmem = LOWMEM_CAP;
```

`INSANE_CUTOFF` 为：

```
#define INSANE_CUTOFF        0x20000U
```

即 128 KB. 上一步我们得到了低地址内存中的低地址部分以及扩展BIOS数据区域，然后调用 `memblock_reserve` 函数来在低内存地址与1MB之间为扩展BIOS数据预留内存区域。

```
lowmem = min(lowmem, ebda_addr);
lowmem = min(lowmem, LOWMEM_CAP);
memblock_reserve(lowmem, 0x100000 - lowmem);
```

`memblock_reserve` 函数定义在 mm/block.c，它接受两个参数：

- 基物理地址
- 区域大小

然后在给定的基地址处预留指定大小的内存。`memblock_reserve` 是在这本书中我们接触到的第一个Linux内核内存管理框架中的函数。我们很快会详细地介绍内存管理，不过现在还是先来看一看这个函数的实现。

Linux内核管理框架初探

在上一段中我们遇到了对 `memblock_reserve` 函数的调用。现在我们来尝试理解一下这个函数是如何工作的。`memblock_reserve` 函数只是调用了：

```
memblock_reserve_region(base, size, MAX_NUMNODES, 0);
```

`memblock_reserve_region` 接受四个参数：

- 内存区域的物理基地址
- 内存区域的大小
- 最大 NUMA 节点数
- 标志参数 flags

在 `memblock_reserve_region` 函数一开始，就是一个 `memblock_type` 结构体类型的变量：

```
struct memblock_type *_rgn = &memblock.reserved;
```

`memblock_type` 类型代表了一块内存，定义如下：

```
struct memblock_type {
         unsigned long cnt;
         unsigned long max;
         phys_addr_t total_size;
         struct memblock_region *regions;
};
```

因为我们要为扩展BIOS数据区域预留内存块，所以当前内存区域的类型就是预留。`memblock` 结构体的定义为：

```
struct memblock {
         bool bottom_up;
         phys_addr_t current_limit;
         struct memblock_type memory;
         struct memblock_type reserved;
#ifdef CONFIG_HAVE_MEMBLOCK_PHYS_MAP
         struct memblock_type physmem;
#endif
};
```

它描述了一块通用的数据块。我们用 `memblock.reserved` 的值来初始化 `_rgn` 。 `memblock` 全局变量定义如下：

```
struct memblock memblock __initdata_memblock = {
    .memory.regions        = memblock_memory_init_regions,
    .memory.cnt        = 1,
    .memory.max        = INIT_MEMBLOCK_REGIONS,
    .reserved.regions    = memblock_reserved_init_regions,
    .reserved.cnt        = 1,
    .reserved.max        = INIT_MEMBLOCK_REGIONS,
#ifdef CONFIG_HAVE_MEMBLOCK_PHYS_MAP
    .physmem.regions    = memblock_physmem_init_regions,
    .physmem.cnt        = 1,
    .physmem.max        = INIT_PHYSMEM_REGIONS,
#endif
    .bottom_up        = false,
    .current_limit        = MEMBLOCK_ALLOC_ANYWHERE,
};
```

我们现在不会继续深究这个变量，但在内存管理部分的中我们会详细地对它进行介绍。需要注意的是，这个变量的声明中使用了 `__initdata_memblock` ：

```
#define __initdata_memblock __meminitdata
```

而 `__meminit_data` 为：

```
#define __meminitdata    __section(.meminit.data)
```

自此我们得出这样的结论：所有的内存块都将定义在 `.meminit.data` 区段中。在我们定义了 `_rgn` 之后，使用了 `memblock_dbg` 宏来输出相关的信息。你可以在从内核命令行传入参数 `memblock=debug` 来开启这些输出。

在输出了这些调试信息后，是对下面这个函数的调用：

```
memblock_add_range(_rgn, base, size, nid, flags);
```

它向 `.meminit.data` 区段添加了一个新的内存块区域。由于 `_rgn` 的值是 `&memblock.reserved` ，下面的代码就直接将扩展BIOS数据区域的基地址、大小和标志填入 `_rgn` 中：

```
if (type->regions[0].size == 0) {
    WARN_ON(type->cnt != 1 || type->total_size);
    type->regions[0].base = base;
    type->regions[0].size = size;
    type->regions[0].flags = flags;
    memblock_set_region_node(&type->regions[0], nid);
    type->total_size = size;
    return 0;
}
```

在填充好了区域后，接着是对 `memblock_set_region_node` 函数的调用。它接受两个参数：

- 填充好的内存区域的地址
- NUMA节点ID

其中我们的区域由 `memblock_region` 结构体来表示：

```
struct memblock_region {
    phys_addr_t base;
    phys_addr_t size;
    unsigned long flags;
#ifdef CONFIG_HAVE_MEMBLOCK_NODE_MAP
    int nid;
#endif
};
```

NUMA节点ID依赖于 `MAX_NUMNODES` 宏，定义在 include/linux/numa.h

```
#define MAX_NUMNODES    (1 << NODES_SHIFT)
```

其中 `NODES_SHIFT` 依赖于 `CONFIG_NODES_SHIFT` 配置参数，定义如下：

```
#ifdef CONFIG_NODES_SHIFT
  #define NODES_SHIFT     CONFIG_NODES_SHIFT
#else
  #define NODES_SHIFT     0
#endif
```

`memblick_set_region_node` 函数只是填充了 `memblock_region` 中的 `nid` 成员：

```
static inline void memblock_set_region_node(struct memblock_region *r, int nid)
{
         r->nid = nid;
}
```

在这之后我们就在 `.meminit.data` 区段拥有了为扩展BIOS数据区域预留的第一个 `memblock` 。`reserve_ebda_region` 已经完成了它该做的任务，我们回到 arch/x86/kernel/head64.c 继续。

至此我们已经结束了进入内核之前所有的准备工作。`x86_64_start_reservations` 的最后一步是调用 init/main.c 中的：

```
start_kernel()
```

这一部分到此结束。

小结

本书的第三部分到这里就结束了。在下一部分中，我们将会见到内核入口点处的初始化工作 —— 位于 `start_kernel` 函数中。这些工作是在启动第一个进程 `init` 之前首先要完成的工作。

如果你有任何问题或建议，请在twitter上联系我 0xAX，或者通过邮件与我沟通，还可以新开issue。

相关链接

- BIOS data area
- What is in the extended BIOS data area on a PC?
- Previous part

内核初始化. Part 4.

Kernel entry point

还记得上一章的内容吗 - 跳转到内核入口之前的最后准备？你应该还记得我们已经完成一系列初始化操作，并停在了调用位于 `init/main.c` 中的 `start_kernel` 函数之前. `start_kernel` 函数是与体系架构无关的通用处理入口函数，尽管我们在此初始化过程中要无数次的返回arch/ 文件夹。如果你仔细看看 `start_kernel` 函数的内容，你将发现此函数涉及内容非常广泛。在此过程中约包含了86个调用函数，是的，你发现它真的是非常庞大但是此部分并不是全部的初始化过程，在当前阶段我们只看这些就可以了。此章节以及后续所有在内核初始化过程章节的内容将涉及并详述它。

`start_kernel` 函数的主要目的是完成内核初始化并启动祖先进程(1号进程)。在祖先进程启动之前 `start_kernel` 函数做了很多事情，如锁验证器,根据处理器标识ID初始化处理器，开启cgroups子系统，设置每CPU区域环境，初始化VFS Cache机制，初始化内存管理，rcu,vmalloc,scheduler(调度器),IRQs(中断向量表),ACPI(中断可编程控制器)以及其它很多子系统。只有经过这些步骤我们才看到本章最后一部分祖先进程启动的过程；同志们，如此复杂的内核子系统，有没有勾起你的学习欲望，有这么多的内核代码等着我们去征服，让我们开始吧。

注意:在此大章节的所有内容 **`Linux Kernel initialization process`** ，并不涉及内核调试相关，关于内核调试部分会有一个单独的章节来进行描述

关于 `__attribute__`

正如我上述所写， `start_kernel` 函数是定义在init/main.c.从已知代码中我们能看到此函数使用了 `__init` 特性，你也许从其它地方了解过关于GCC `__attribute__` 相关的内容。在内核初始化阶段这个机制在所有的函数中都是有必要的。

```
#define __init      __section(.init.text) __cold notrace
```

在初始化过程完成后，内核将通过调用 `free_initmem` 释放这些sections(段)。注意 `__init` 属性是通过 `__cold` 和 `notrace` 两个属性来定义的。第一个属性 `cold` 的目的是标记此函数很少使用所以编译器必须优化此函数的大小，第二个属性 `notrace` 定义如下：

```
#define notrace __attribute__((no_instrument_function))
```

含有 `no_instrument_function` 意思就是告诉编译器函数调用不产生环境变量(堆栈空间)。

在 `start_kernel` 函数的定义中，你也可以看到 `__visible` 属性的扩展：

```
#define __visible __attribute__((externally_visible))
```

含有 `externally_visible` 意思就是告诉编译器有一些过程在使用该函数或者变量，为了放至标记这个函数/变量是 `unusable` 。你可以在此include/linux/init.h处查到这些属性表达式的含义。

start_kernel 初始化

在start_kernel的初始之初你可以看到这两个变量：

```
char *command_line;
char *after_dashes;
```

第一个变量表示内核命令行的全局指针，第二个变量将包含 `parse_args` 函数通过输入字符串中的参数'name=value'，寻找特定的关键字和调用正确的处理程序。我们不想在这个时候参与这两个变量的相关细节，但是会在接下来的章节看到。我们接着往下走，下一步我们看到了此函数:

```
lockdep_init();
```

`lockdep_init` 初始化 lock validator. 其实现是相当简单的，它只是初始化了两个哈希表 list_head并设置 `lockdep_initialized` 全局变量为 `1` 。 关于自旋锁 spinlock以及互斥锁mutex 如何获取请参考链接.

下一个函数是 `set_task_stack_end_magic` ，参数为 `init_task` 和设置 `STACK_END_MAGIC` (`0x57AC6E9D`)。 `init_task` 代表初始化进程(任务)数据结构:

```
struct task_struct init_task = INIT_TASK(init_task);
```

`task_struct` 存储了进程的所有相关信息。因为它很庞大，我在这本书并不会去介绍，详细信息你可以查看调度相关数据结构定义头文件 include/linux/sched.h。在此刻 `task_sreuct` 包含了超过 `100` 个字段！虽然你不会看到 `task_struct` 是在这本书中的解释，但是我们会经常使用它，因为它是介绍在Linux内核 `进程` 的基本知识。我将描述这个结构中字段的一些含义，因为我们在后面的实践中见到它们。

你也可以查看 `init_task` 的相关定义以及宏指令 `INIT_TASK` 的初始化流程。这个宏指令来自于include/linux/init_task.h在此刻只是设置和初始化了第一个进程来(0号进程)的值。例如这么设置：

- 初始化进程状态为 zero 或者 `runnable` . 一个可运行进程即为等待CPU去运行;
- 初始化仅存的标志位 - `PF_KTHREAD` 意思为 - 内核线程;
- 一个可运行的任务列表;
- 进程地址空间;
- 初始化进程堆栈 `&init_thread_info` - `init_thread_union.thread_info` 和 `initthread_union` 使用共用体 - `thread_union` 包含了 `thread_info` 进程信息以及进程栈:。

```
union thread_union {
    struct thread_info thread_info;
    unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

每个进程都有其自己的堆栈， `x86_64` 架构的CPU一般支持的页表是16KB or 4个页框大小。我们注意stack变量被定义为数据并且类型是 `unsigned long` 。 `thread_union` 结构的下一个字段为 `thread_union` 定义如下：

```
struct thread_info {
        struct task_struct      *task;
        struct exec_domain      *exec_domain;
        __u32                   flags;
        __u32                   status;
        __u32                   cpu;
        int                     saved_preempt_count;
        mm_segment_t            addr_limit;
        struct restart_block    restart_block;
        void __user             *sysenter_return;
        unsigned int            sig_on_uaccess_error:1;
        unsigned int            uaccess_err:1;
};
```

此结构占用52个字节。 `thread_info` 结构包含了特定体系架构相关的线程信息，我们都知道在 `X86_64` 架构上内核栈是逆生成而 `thread_union.thread_info` 结构则是正生长。所以进程进程栈是16KB并且 `thread_info` 是在栈底。还需我们处理 `16 kilobytes - 62 bytes = 16332 bytes` .注意 `thread_union` 代表一个联合体union而不是结构体，用一张图来描述栈内存空间。 如下图所示:

```
+-----------------------+
|                       |
|                       |
|         stack         |
|                       |
|_______________________|
|          |            |
|          |            |
|          |            |
|__________↓____________|             +--------------------+
|                       |             |                    |
|      thread_info      |<----------->|     task_struct    |
|                       |             |                    |
+-----------------------+             +--------------------+
```

http://www.quora.com/In-Linux-kernel-Why-thread_info-structure-and-the-kernel-stack-of-a-process-binds-in-union-construct

所以 `INIT_TASK` 宏指令就是 `task_struct's` '结构。正如我上述所写，我并不会去描述这些字段的含义和值，在 `INIT_TASK` 赋值处理的时候我们很快能看到这些。

现在让我们回到 `set_task_stack_end_magic` 函数，这个函数被定义在kernel/fork.c功能为设置canary `init` 进程堆栈以检测堆栈溢出。

```
void set_task_stack_end_magic(struct task_struct *tsk)
{
    unsigned long *stackend;
    stackend = end_of_stack(tsk);
    *stackend = STACK_END_MAGIC; /* for overflow detection */
}
```

上述函数比较简单，`set_task_stack_end_magic` 函数的作用是先通过 `end_of_stack` 函数获取堆栈并赋给 `task_struct` 。关于检测配置需要打开内核配置宏 `CONFIG_STACK_GROWSUP` 。因为我们学习的是x86架构的初始化，堆栈是逆生成，所以堆栈底部为：

```
(unsigned long *)(task_thread_info(p) + 1);
```

`task_thread_info` 的定义如下，返回一个当前的堆栈；

```
#define task_thread_info(task)  ((struct thread_info *)(task)->stack)
```

进程的栈底，我们写 `STACK_END_MAGIC` 这个值。如果设置 `canary` ，我们可以像这样子去检测堆栈：

```
if (*end_of_stack(task) != STACK_END_MAGIC) {
        //
        // handle stack overflow here
        //
}
```

`set_task_stack_end_magic` 初始化完毕后的下一个函数是 `smp_setup_processor_id` .此函数在 `x86_64` 架构上是空函数：

```
void __init __weak smp_setup_processor_id(void)
{
}
```

在此架构上没有实现此函数，但在别的体系架构的实现可以参考s390 and arm64.

我们接着往下走，下一个函数是 `debug_objects_early_init` 。此函数的执行几乎和 `lockdep_init` 是一样的，但是填充的哈希对象是调试相关。上述我已经表明，关于内核调试部分会在后续专门有一个章节来完成。

`debug_object_early_init` 函数之后我们看到调用了 `boot_init_stack_canary` 函数。 `task_struct->canary` 的值利用了GCC特性，但是此特性需要先使能内核 `CONFIG_CC_STACKPROTECTOR` 宏后才可以使用。 `boot_init_stack_canary` 什么也没有做， 否则基于随机数和随机池产生 TSC:

```
get_random_bytes(&canary, sizeof(canary));
tsc = __native_read_tsc();
canary += tsc + (tsc << 32UL);
```

我们要获取随机数, 我们可以给 `stack_canary` 字段 `task_struct` 赋值：

```
current->stack_canary = canary;
```

然后将此值写入IRQ堆栈的顶部:

```
this_cpu_write(irq_stack_union.stack_canary, canary); // read below about this_cpu_write
```

关于IRQ的章节我们这里也不会详细刨析, 关于这部分介绍看这里IRQs.如果canary被设置, 关闭本地中断注册bootstrap CPU以及CPU maps. 我们关闭本地中断 (interrupts for current CPU) 使用 `local_irq_disable` 函数，展开后原型为 `arch_local_irq_disable` 函数include/linux/percpu-defs.h:

```
static inline notrace void arch_local_irq_enable(void)
{
        native_irq_enable();
}
```

如果 `native_irq_enable` 通过 `cli` 指令判断架构，这里是 `X86_64` ， Where `native_irq_enable` is `cli` instruction for `x86_64` .中断的关闭(屏蔽)我们可以通过注册当前CPU ID到CPU bitmap来实现。

激活第一个CPU

当前已经走到 `start_kernel` 函数中的 `boot_cpu_init` 函数，此函数主要为了通过掩码初始化每一个CPU。首先我们需要获取当前处理器的ID通过下面函数：

```
int cpu = smp_processor_id();
```

现在是0. 如果 `CONFIG_DEBUG_PREEMPT` 宏配置了那么 `smp_processor_id` 的值就来自于 `raw_smp_processor_id` 函数，原型如下:

```
#define raw_smp_processor_id() (this_cpu_read(cpu_number))
```

`this_cpu_read` 函数与其它很多函数一样如(`this_cpu_write` , `this_cpu_add` 等等...) 被定义在include/linux/percpu-defs.h 此部分函数主要为对 `this_cpu` 进行操作. 这些操作提供不同的对每cpuper-cpu 变量相关访问方式. 譬如让我们来看看这个函数 `this_cpu_read` :

```
__pcpu_size_call_return(this_cpu_read_, pcp)
```

还记得上面我们所写，每cpu变量 `cpu_number` 的值是 `this_cpu_read` 通过 `raw_smp_processor_id` 来得到，现在让我们看看 `__pcpu_size_call_return` 的执行：

```
#define __pcpu_size_call_return(stem, variable)                        \
({                                                                      \
        typeof(variable) pscr_ret__;                                    \
        __verify_pcpu_ptr(&(variable));                                 \
        switch(sizeof(variable)) {                                      \
        case 1: pscr_ret__ = stem##1(variable); break;                  \
        case 2: pscr_ret__ = stem##2(variable); break;                  \
        case 4: pscr_ret__ = stem##4(variable); break;                  \
        case 8: pscr_ret__ = stem##8(variable); break;                  \
        default:                                                        \
                __bad_size_call_parameter(); break;                     \
        }                                                               \
        pscr_ret__;                                                     \
})
```

是的，此函数虽然看起起奇怪但是它的实现是简单的，我们看到 `pscr_ret__` 变量的定义是 `int` 类型，为什么是int类型呢？好吧， `变量` 是 `common_cpu` 它声明了每cpu(per-cpu)变量:

```
DECLARE_PER_CPU_READ_MOSTLY(int, cpu_number);
```

在下一个步骤中我们调用了 `__verify_pcpu_ptr` 通过使用一个有效的每cpu变量指针来取地址得到 `cpu_number` 。之后我们通过 `pscr_ret__` 函数设置变量的大小， `common_cpu` 变量是 `int` ,所以它的大小是4字节。意思就是我们通过 `this_cpu_read_4(common_cpu)` 获取cpu变量其大小被 `pscr_ret__` 决定。在 `__pcpu_size_call_return` 的结束 我们调用了 __pcpu_size_call_return：

```
#define this_cpu_read_4(pcp)       percpu_from_op("mov", pcp)
```

需要调用 `percpu_from_op` 并且通过 `mov` 指令来传递每cpu变量， `percpu_from_op` 的内联扩展如下：

```
asm("movl %%gs:%1,%0" : "=r" (pfo_ret__) : "m" (common_cpu))
```

让我们尝试理解此函数是如果工作的， `gs` 段寄存器包含每个CPU区域的初始值，这里我们通过 `mov` 指令copy `common_cpu` 到内存中去，此函数还有另外的形式：

```
this_cpu_read(common_cpu)
```

等价于:

```
movl %gs:$common_cpu, $pfo_ret__
```

由于我们没有设置每个CPU的区域,我们只有一个 - 为当前CPU的值 `zero` 通过此函数 `smp_processor_id` 返回.

返回的ID表示我们处于哪一个CPU上, `boot_cpu_init` 函数设置了CPU的在线, 激活, 当前的设置为:

```
set_cpu_online(cpu, true);
set_cpu_active(cpu, true);
set_cpu_present(cpu, true);
set_cpu_possible(cpu, true);
```

上述我们所有使用的这些CPU的配置我们称之为- CPU掩码 `cpumask` . `cpu_possible` 则是设置支持CPU热插拔时候的CPU ID. `cpu_present` 表示当前热插拔的CPU. `cpu_online` 表示当前所有在线的CPU以及通过 `cpu_present` 来决定被调度出去的CPU. CPU热插拔的操作需要打开内核配置宏 `CONFIG_HOTPLUG_CPU` 并且将 `possible == present` 以及 `active == online` 选项禁用。这些功能都非常相似，每个函数都需要检查第二个参数，如果设置为 `true` ，需要通过调用 `cpumask_set_cpu` or `cpumask_clear_cpu` 来改变状态。

譬如我们可以通过true或者第二个参数来这么调用：

```
cpumask_set_cpu(cpu, to_cpumask(cpu_possible_bits));
```

让我们继续尝试理解 `to_cpumask` 宏指令，此宏指令转化为一个位图通过 `struct cpumask *` ，CPU掩码提供了位图集代表了当前系统中所有的CPU's，每CPU都占用1bit，CPU掩码相关定义通过 `cpu_mask` 结构定义:

```
typedef struct cpumask { DECLARE_BITMAP(bits, NR_CPUS); } cpumask_t;
```

在来看下面一组函数定义了位图宏指令。

```
#define DECLARE_BITMAP(name, bits) unsigned long name[BITS_TO_LONGS(bits)]
```

正如我们看到的定义一样， `DECLARE_BITMAP` 宏指令的原型是一个 `unsigned long` 的数组，现在让我们查看如何执行 `to_cpumask` ：

```
#define to_cpumask(bitmap)                                          \
        ((struct cpumask *)(1 ? (bitmap)                            \
                            : (void *)sizeof(__check_is_bitmap(bitmap))))
```

我不知道你是怎么想的, 但是我是这么想的，我看到此函数其实就是一个条件判断语句当条件为真的时候，但是为什么执行 `__check_is_bitmap` ？让我们看看 `__check_is_bitmap` 的定义：

```
static inline int __check_is_bitmap(const unsigned long *bitmap)
{
        return 1;
}
```

原来此函数始终返回1，事实上我们需要这样的函数才达到我们的目的： 它在编译时给定一个 `bitmap` ，换句话将就是检查 `bitmap` 的类型是否是 `unsigned long *` ,因此我们仅仅通过 `to_cpumask` 宏指令将类型为 `unsigned long` 的数组转化为 `struct cpumask *` 。现在我们可以调用 `cpumask_set_cpu` 函数，这个函数仅仅是一个 `set_bit` 给CPU掩码的功能函数。所有的这些 `set_cpu_*` 函数的原理都是一样的。

如果你还不确定 `set_cpu_*` 这些函数的操作并且不能理解 `cpumask` 的概念，不要担心。你可以通过读取这些章节cpumask or documentation.来继续了解和学习这些函数的原理。

现在我们已经激活第一个CPU，我们继续接着start_kernel函数往下走，下面的函数是 `page_address_init` ,但是此函数不执行任何操作，因为只有当所有内存不能直接映射的时候才会执行。

Linux 内核的第一条打印信息

下面调用了pr_notice函数。

```
#define pr_notice(fmt, ...) \
    printk(KERN_NOTICE pr_fmt(fmt), ##__VA_ARGS__)
```

pr_notice其实是printk的扩展，这里我们使用它打印了Linux 的banner。

```
pr_notice("%s", linux_banner);
```

打印的是内核的版本号以及编译环境信息:

```
Linux version 4.0.0-rc6+ (alex@localhost) (gcc version 4.9.1 (Ubuntu 4.9.1-16ubuntu6)
) #319 SMP
```

依赖于体系结构的初始化部分

下个步骤我们就要进入到指定的体系架构的初始函数，Linux 内核初始化体系架构相关调用 `setup_arch` 函数，这又是一个类型于 `start_kernel` 的庞大函数，这里我们仅仅简单描述，在下一个章节我们将继续深入。指定体系架构的内容，我们需要再一次阅读 `arch/` 目

录，`setup_arch` 函数定义在arch/x86/kernel/setup.c 文件中，此函数就一个参数-内核命令行。

此函数解析内核的段 `_text` 和 `_data` 来自于 `_text` 符号和 `_bss_stop` (你应该还记得此文件 arch/x86/kernel/head_64.S)。我们使用 `memblock` 来解析内存块。

```
memblock_reserve(__pa_symbol(_text), (unsigned long)__bss_stop - (unsigned long)_text)
;
```

你可以阅读关于 `memblock` 的相关内容在Linux kernel memory management Part 1.，你应该还记得 `memblock_reserve` 函数的两个参数：

- base physical address of a memory block;
- size of a memory block.

我们可以通过 `__pa_symbol` 宏指令来获取符号表 `_text` 段中的物理地址

```
#define __pa_symbol(x) \
    __phys_addr_symbol(__phys_reloc_hide((unsigned long)(x)))
```

上述宏指令调用 `__phys_reloc_hide` 宏指令来填充参数，`__phys_reloc_hide` 宏指令在 `x86_64` 上返回的参数是给定的。宏指令 `__phys_addr_symbol` 的执行是简单的，只是减去从 `_text` 符号表中读到的内核的符号映射地址并且加上物理地址的基地址。

```
#define __phys_addr_symbol(x) \
 ((unsigned long)(x) - __START_KERNEL_map + phys_base)
```

`memblock_reserve` 函数对内存页进行分配。

保留可用内存初始化initrd

之后我们保留替换内核的text和data段用来初始化initrd,我们暂时不去了解initrd的详细信息，你仅仅只需要知道根文件系统就是通过这方式来进行初始化这就是 `early_reserve_initrd` 函数的工作，此函数获取RAM DISK的基地址以及大小以及大小加偏移。

```
u64 ramdisk_image = get_ramdisk_image();
u64 ramdisk_size  = get_ramdisk_size();
u64 ramdisk_end   = PAGE_ALIGN(ramdisk_image + ramdisk_size);
```

如果你阅读过这些章节Linux Kernel Booting Process，你就知道所有的这些啊参数都来自于 `boot_params` ，时刻谨记 `boot_params` 在boot期间已经被赋值，内核启动头包含了一下几个字段用来描述RAM DISK：

```
Field name:     ramdisk_image
Type:         write (obligatory)
Offset/size:    0x218/4
Protocol:     2.00+

  The 32-bit linear address of the initial ramdisk or ramfs.  Leave at
  zero if there is no initial ramdisk/ramfs.
```

我们可以得到关于 `boot_params` 的一些信息. 具体查看 `get_ramdisk_image`：

```
static u64 __init get_ramdisk_image(void)
{
        u64 ramdisk_image = boot_params.hdr.ramdisk_image;

        ramdisk_image |= (u64)boot_params.ext_ramdisk_image << 32;

        return ramdisk_image;
}
```

关于32位的ramdisk的地址，我们可以阅读此部分内容来获取Documentation/x86/zero-page.txt:

```
0C0/004    ALL    ext_ramdisk_image ramdisk_image high 32bits
```

32位变化后，我们获取64位的ramdisk原理一样，为此我们可以检查bootloader 提供的ramdisk信息：

```
if (!boot_params.hdr.type_of_loader ||
    !ramdisk_image || !ramdisk_size)
    return;
```

并保留内存块将ramdisk传输到最终的内存地址，然后进行初始化：

```
memblock_reserve(ramdisk_image, ramdisk_end - ramdisk_image);
```

结束语

以上就是第四部分关于内核初始化的部分内容，我们从 `start_kernel` 函数开始一直到指定体系架构初始化 `setup_arch` 的过程中停止，那么在下一个章节我们将继续研究体系架构相关的初始化内容。

如果你有任何的问题或者建议，你可以留言，也可以直接发消息给我twitter。

很抱歉，英语并不是我的母的，非常抱歉给您阅读带来不便，如果你发现文中描述有任何问题，请提交一个 **PR** 到 **linux-insides**.

链接

* GCC function attributes
* this_cpu operations
* cpumask
* lock validator
* cgroups
* stack buffer overflow
* IRQs
* initrd
* Previous part

内核初始化 第五部分

与系统架构有关的初始化后续分析

在之前的章节中， 我们讲到了与系统架构有关的 setup_arch 函数部分，本文会继续从这里开始。 我们为 initrd 预留了内存之后，下一步是执行 `olpc_ofw_detect` 函数检测系统是否支持 One Laptop Per Child support。 我们不会考虑与平台有关的东西，且会忽略与平台有关的函数。所以我们继续往下看。 下一步是执行 `early_trap_init` 函数。这个函数会初始化调试功能 （ `#DB` -当 `TF` 标志位和rflags被设置时会被使用）和 `int3` （ `#BP` ）中断门。 如果你不了解中断，你可以从 初期中断和异常处理 中学习有关中断的内容。 在 `x86` 架构中， `INT` ， `INT0` 和 `INT3` 是支持任务显式调用中断处理函数的特殊指令。 `INT3` 指令调用断点（ `#BP` ）处理函数。 你如果记得，我们在这部分 看到过中断和异常概念：

```
----------------------------------------------------------------------------------------
|Vector|Mnemonic|Description         |Type |Error Code|Source                   |
----------------------------------------------------------------------------------------
|3     | #BP    |Breakpoint          |Trap |NO        |INT 3                    |
----------------------------------------------------------------------------------------
```

调试中断 `#DB` 是激活调试器的重要方法。 `early_trap_init` 函数的定义在 arch/x86/kernel/traps.c 中。这个函数用来设置 `#DB` 和 `#BP` 处理函数，并且实现重新加载 IDT。

```c
void __init early_trap_init(void)
{
        set_intr_gate_ist(X86_TRAP_DB, &debug, DEBUG_STACK);
        set_system_intr_gate_ist(X86_TRAP_BP, &int3, DEBUG_STACK);
        load_idt(&idt_descr);
}
```

我们之前中断相关章节中看到过 `set_intr_gate` 的实现。这里的 `set_intr_gate_ist` 和 `set_system_intr_gate_ist` 也是类似的实现。 这两个函数都需要三个参数：

- 中断号
- 中断/异常处理函数的基地址
- 第三个参数是 `Interrupt Stack Table` 。 `IST` 是 TSS 的部分内容，是 `x86_64` 引入的

> 新机制。在内核态处于活跃状态的线程拥有 `16kb` 的内核栈空间。但是在用户空间的线程的内核栈是空的。除了线程栈，还有一些与每个 `CPU` 有关的特殊栈。你可以查阅 linux 内核文档 - Kernel stacks 部分了解这些栈信息。 `x86_64` 提供了像在非屏蔽中断等类似事件中切换新的特殊栈的特性支持。这个特性的名字是 `Interrupt Stack Table` 。每个CPU最多可以有 7 个 `IST` 条目，每个条目有自己特定的栈。在我们的案例中使用的是 `DEBUG_STACK` 。

`set_intr_gate_ist` 和 `set_system_intr_gate_ist` 与 `set_intr_gate` 的工作原理几乎一样，只有一个区别。 这些函数检查中断号并在内部调用 `_set_gate` ：

```
BUG_ON((unsigned)n > 0xFF);
_set_gate(n, GATE_INTERRUPT, addr, 0, ist, __KERNEL_CS);
```

其中， `set_intr_gate` 把 dpl 和 `ist` 置为 0 来调用 `_set_gate` 。 但是 `set_intr_gate_ist` 和 `set_system_intr_gate_ist` 把 `ist` 设置为 `DEBUG_STACK` ，并且 `set_system_intr_gate_ist` 把 `dpl` 设置为优先级最低的 `0x3` 。 当中断发生时，硬件加载这个描述符，然后硬件根据 `IST` 的值自动设置新的栈指针。 之后激活对应的中断处理函数。所有的特殊内核栈会在 `cpu_init` 函数中设置好（我们会在后文中提到）。

当 `#DB` 和 `#BP` 门向 `idt_descr` 有写操作，我们会调用 `load_idt` 函数来执行 `ldtr` 指令来重新加载 `IDT` 表。 现在我们来了解下中断处理函数并尝试理解它的工作原理。当然，我们不可能在这本书中讲解所有的中断处理函数。 深入学习 linux 的内核源码是很有意思的事情，我们会在这里讲解 `debug` 处理函数的实现。请自行学习其他的中断处理函数实现。

`#DB` 处理函数

像上文中提到的，我们在 `set_intr_gate_ist` 中通过 `&debug` 的地址传送 `#DB` 处理函数。lxr.free-electorns.com 是很好的用来搜索 linux 源代码中标识符的资源。 遗憾的是，你在其中找不到 `debug` 处理函数。你只能在 arch/x86/include/asm/traps.h 中找到 `debug` 的定义：

```
asmlinkage void debug(void);
```

从 `asmlinkage` 属性我们可以知道 `debug` 是由 assembly 语言实现的函数。是的，又是汇编语言 :)。 和其他处理函数一样， `#DB` 处理函数的实现可以在 arch/x86/kernel/entry_64.S 文件中找到。 都是由 `idtentry` 汇编宏定义的：

```
idtentry debug do_debug has_error_code=0 paranoid=1 shift_ist=DEBUG_STACK
```

`idtentry` 是一个定义中断/异常指令入口点的宏。它需要五个参数：

- 中断条目点的名字
- 中断处理函数的名字
- 是否有中断错误码
- paranoid - 如果这个参数置为 1，则切换到特殊栈
- shift_ist - 支持中断期间切换栈

现在我们来看下 `idtentry` 宏的实现。这个宏的定义也在相同的汇编文件中，并且定义了有 `ENTRY` 宏属性的 `debug` 函数。 首先，`idtentry` 宏检查所有的参数是否正确，是否需要切换到特殊栈。接下来检查中断返回的错误码。例如本案例中的 `#DB` 不会返回错误码。 如果有错误码返回，它会调用 `INTR_FRAME` 或者 `XCPT_FRAM` 宏。其实 `XCPT_FRAME` 和 `INTR_FRAME` 宏什么也不会做，只是对中断初始状态编译的时候有用。 它们使用 `CFI` 指令用来调试。你可以查阅更多有关 `CFI` 指令的信息 CFI。 就像 arch/x86/kernel/entry_64.S 中解释： `CFI` 宏是用来产生更好的回溯的 `dwarf2` 的解开信息。 它们不会改变任何代码。因此我们可以忽略它们。

```
.macro idtentry sym do_sym has_error_code:req paranoid=0 shift_ist=-1
ENTRY(\sym)
    /* Sanity check */
    .if \shift_ist != -1 && \paranoid == 0
    .error "using shift_ist requires paranoid=1"
    .endif

    .if \has_error_code
    XCPT_FRAME
    .else
    INTR_FRAME
    .endif
    ...
    ...
    ...
```

当中断发生后经过初期的中断/异常处理，我们可以知道栈内的格式是这样的：

```
    +-----------------------+
    |                       |
+40 |         SS            |
+32 |         RSP           |
+24 |        RFLAGS         |
+16 |         CS            |
+8  |         RIP           |
 0  |       Error Code      | <---- rsp
    |                       |
    +-----------------------+
```

`idtentry` 实现中的另外两个宏分别是

```
    ASM_CLAC
    PARAVIRT_ADJUST_EXCEPTION_FRAME
```

第一个 `ASM_CLAC` 宏依赖于 `CONFIG_X86_SMAP` 这个配置项和考虑安全因素，你可以从这里了解更多内容。 第二个 `PARAVIRT_EXCEPTION_FRAME` 宏是用来处理 `Xen` 类型异常（这章只讲解内核初始化，不会考虑虚拟化的内容）。 下一段代码会检查中断是否有错误码。如果没有则会把 `$-1` (在 `x86_64` 架构下值为 `0xffffffffffffffff`)压入栈：

```
    .ifeq \has_error_code
    pushq_cfi $-1
    .endif
```

为了保证对于所有中断的栈的一致性，我们会把它处理为 `dummy` 错误码。下一步我们从栈指针中减去 `$ORIG_RAX-R15` ：

```
    subq $ORIG_RAX-R15, %rsp
```

其中， `ORIG_RAX` ， `R15` 和其他宏都定义在 arch/x86/include/asm/calling.h 中。 `ORIG_RAX-R15` 是 120 字节。 我们在中断处理过程中需要把所有的寄存器信息存储在栈中，所有通用寄存器会占用这个 120 字节。 为通用寄存器设置完栈之后，下一步是检查从用户空间产生的中断：

```
testl $3, CS(%rsp)
jnz 1f
```

我们查看段寄存器 `CS` 的前两个比特位。你应该记得 `CS` 寄存器包含段选择器，它的前两个比特是 `RPL` 。所有的权限等级是0-3范围内的整数。 数字越小代表权限越高。因此当中断来自内核空间，我们会调用 `save_paranoid` ，如果不来自内核空间，我们会跳转到标签 `1` 处处理。 在 `save_paranoid` 函数中，我们会把所有的通用寄存器存储到栈中，如果需要的话会用户态 `gs` 切换到内核态 `gs` ：

```
movl $1,%ebx
    movl $MSR_GS_BASE,%ecx
    rdmsr
    testl %edx,%edx
    js 1f
    SWAPGS
    xorl %ebx,%ebx
1:    ret
```

下一步我们把 `pt_regs` 指针存在 `rdi` 中，如果存在错误码就把它存储到 `rsi` 中，然后调用中断处理函数，例如就像 arch/x86/kernel/trap.c中的 `do_debug` 。 `do_debug` 像其他处理函数一样需要两个参数：

- pt_regs - 是一个存储在进程内存区域的一组CPU寄存器
- error code - 中断错误码

中断处理函数完成工作后会调用 `paranoid_exit` 还原栈区。如果中断来自用户空间则切换回用户态并调用 `iret` 。我们会在不同的章节继续深入分析中断。 这是用在 `#DB` 中断中的 `idtentry` 宏的基本介绍。所有的中断都和这个实现类似，都定义在 `idtentry` 中。 `early_trap_init` 执行完后，下一个函数是 `early_cpu_init` 。 这个函数定义在 arch/x86/kernel/cpu/common.c 中，负责收集 `CPU` 和其供应商的信息。

早期ioremap初始化

下一步是初始化早期的 `ioremap` 。通常有两种实现与设备通信的方式：

- I/O端口
- 设备内存

我们在 linux 内核启动过程中见过第一种方法（通过 `outb/inb` 指令实现）。 第二种方法是把 `I/O` 的物理地址映射到虚拟地址。当 `CPU` 读取一段物理地址时，它可以读取到映射了 `I/O` 设备的物理 `RAM` 区域。 `ioremap` 就是用来把设备内存映射到内核地址空间的。

像我上面提到的下一个函数时 `early_ioremap_init` ，它可以在正常的像 `ioremap` 这样的映射函数可用之前，把 `I/O` 内存映射到内核地址空间以方便读取。 我们需要在初期的初始化代码中初始化临时的 `ioremap` 来映射 `I/O` 设备到内存区域。初期的 `ioremap` 实现在 arch/x86/mm/ioremap.c 中可以找到。 在 `early_ioremap_init` 的一开始我们可以看到 `pmd_t` 类型的 `pmd` 指针定义（代表页中间目录条目 `typedef struct {pmdval_t pmd; } pmd_t;` 其中 `pmdval_t` 是无符号长整型）。 然后检查 `fixmap` 是正确对齐的：

```
pmd_t *pmd;
BUILD_BUG_ON((fix_to_virt(0) + PAGE_SIZE) & ((1 << PMD_SHIFT) - 1));
```

`fixmap` - 是一段从 `FIXADDR_START` 到 `FIXADDR_TOP` 的固定虚拟地址映射区域。它在子系统需要知道虚拟地址的编译过程中会被使用。 之后 `early_ioremap_init` 函数会调用 mm/early_ioremap.c 中的 `early_ioremap_setup` 函数。 `early_ioremap_setup` 会填充512个临时的启动时固定映射表来完成无符号长整型矩阵 `slot_virt` 的初始化：

```
for (i = 0; i < FIX_BTMAPS_SLOTS; i++)
    slot_virt[i] = __fix_to_virt(FIX_BTMAP_BEGIN - NR_FIX_BTMAPS*i);
```

之后我们就获得了 `FIX_BTMAP_BEGIN` 的页中间目录条目，并把它赋值给了 `pmd` 变量，把启动时间页表 `bm_pte` 写满 0。然后调用 `pmd_populate_kernel` 函数设置给定的页中间目录的页表条目：

```
pmd = early_ioremap_pmd(fix_to_virt(FIX_BTMAP_BEGIN));
memset(bm_pte, 0, sizeof(bm_pte));
pmd_populate_kernel(&init_mm, pmd, bm_pte);
```

这就是所有过程。如果你仍然觉得困惑，不要担心。在 内核内存管理，第二部分 章节会有单独一部分讲解 `ioremap` 和 `fixmaps` 。

获取根设备的主次设备号

`ioremap` 初始化完成后，紧接着是执行下面的代码：

```
ROOT_DEV = old_decode_dev(boot_params.hdr.root_dev);
```

这段代码用来获取根设备的主次设备号。后面 `initrd` 会通过 `do_mount_root` 函数挂载到这个根设备上。其中主设备号用来识别和这个设备有关的驱动。 次设备号用来表示使用该驱动的各设备。注意 `old_decode_dev` 函数是从 `boot_params_structure` 中获取了一个参数。我们可以从 x86 linux 内核启动协议中查到：

```
Field name:    root_dev
Type:        modify (optional)
Offset/size:    0x1fc/2
Protocol:     ALL

  The default root device device number.  The use of this field is
  deprecated, use the "root=" option on the command line instead
```

现在我们来看看 `old_decode_dev` 如何实现的。实际上它只是根据主次设备号调用了 `MKDEV` 来生成一个 `dev_t` 类型的设备。它的实现很简单：

```
static inline dev_t old_decode_dev(u16 val)
{
         return MKDEV((val >> 8) & 255, val & 255);
}
```

其中 `dev_t` 是用来表示主/次设备号对的一个内核数据类型。但是这个奇怪的 `old` 前缀代表了什么呢？出于历史原因，有两种管理主次设备号的方法。 第一种方法主次设备号占用 2 字节。你可以在以前的代码中发现：主设备号占用 8 bit，次设备号占用 8 bit。但是这会引入一

个问题：最多只能支持 256 个主设备号和 256 个次设备号。 因此后来引入了 32 bit 来表示主次设备号，其中 12 位用来表示主设备号，20 位用来表示次设备号。你可以在 `new_decode_dev` 的实现中找到：

```
static inline dev_t new_decode_dev(u32 dev)
{
        unsigned major = (dev & 0xfff00) >> 8;
        unsigned minor = (dev & 0xff) | ((dev >> 12) & 0xfff00);
        return MKDEV(major, minor);
}
```

如果 `dev` 的值是 `0xffffffff` ，经过计算我们可以得到用来表示主设备号的 12 位值 `0xfff` ，表示次设备号的20位值 `0xfffff` 。因此经过 `old_decode_dev` 我们最终可以得到在 `ROOT_DEV` 中根设备的主次设备号。

Memory Map设置

下一步是调用 `setup_memory_map` 函数设置内存映射。但是在这之前我们需要设置与显示屏有关的参数（目前有行、列，视频页等，你可以在 显示模式初始化和进入保护模式 中了解），与拓展显示识别数据，视频模式，引导启动器类型等参数：

```
screen_info = boot_params.screen_info;
    edid_info = boot_params.edid_info;
    saved_video_mode = boot_params.hdr.vid_mode;
    bootloader_type = boot_params.hdr.type_of_loader;
    if ((bootloader_type >> 4) == 0xe) {
        bootloader_type &= 0xf;
        bootloader_type |= (boot_params.hdr.ext_loader_type+0x10) << 4;
    }
    bootloader_version  = bootloader_type & 0xf;
    bootloader_version |= boot_params.hdr.ext_loader_ver << 4;
```

我们可以从启动时候存储在 `boot_params` 结构中获取这些参数信息。之后我们需要设置 `I/O` 内存。众所周知，内核主要做的工作就是资源管理。其中一个资源就是内存。 我们也知道目前有通过 `I/O` 口和设备内存两种方法实现设备通信。所有有关注册资源的信息可以通过 `/proc/ioports` 和 `/proc/iomem` 获得：

- /proc/ioports - 提供用于设备输入输出通信的一组注册端口区域
- /proc/iomem - 提供每个物理设备的系统内存映射地址 我们先来看下 `/proc/iomem` ：

```
cat /proc/iomem
00000000-00000fff : reserved
00001000-0009d7ff : System RAM
0009d800-0009ffff : reserved
000a0000-000bffff : PCI Bus 0000:00
000c0000-000cffff : Video ROM
000d0000-000d3fff : PCI Bus 0000:00
000d4000-000d7fff : PCI Bus 0000:00
000d8000-000dbfff : PCI Bus 0000:00
000dc000-000dffff : PCI Bus 0000:00
000e0000-000fffff : reserved
  000e0000-000e3fff : PCI Bus 0000:00
  000e4000-000e7fff : PCI Bus 0000:00
  000f0000-000fffff : System ROM
```

可以看到，根据不同属性划分为以十六进制符号表示的一段地址范围。linux 内核提供了用来管理所有资源的一种通用 API。全局资源（比如 PICs 或者 I/O 端口）可以划分为与硬件总线插槽有关的子集。 `resource` 的主要结构是：

```
struct resource {
        resource_size_t start;
        resource_size_t end;
        const char *name;
        unsigned long flags;
        struct resource *parent, *sibling, *child;
};
```

例如下图中的树形系统资源子集示例。这个结构提供了资源占用的从 `start` 到 `end` 的地址范围（ `resource_size_t` 是 `phys_addr_t` 类型，在 `x86_64` 架构上是 `u64` ）。资源名（你可以在 `/proc/iomem` 输出中看到），资源标记（所有的资源标记定义在 include/linux/ioport.h 文件中）。最后三个是资源结构体指针，如下图所示：

```
+-------------+      +-------------+
|             |      |             |
|    parent   |------|    sibling  |
|             |      |             |
+-------------+      +-------------+
       |
       |
+-------------+
|             |
|    child    |
|             |
+-------------+
```

每个资源子集有自己的根范围资源。 `iomem` 的资源 `iomem_resource` 的定义是：

```
struct resource iomem_resource = {
        .name   = "PCI mem",
        .start  = 0,
        .end    = -1,
        .flags  = IORESOURCE_MEM,
};
EXPORT_SYMBOL(iomem_resource);
TODO EXPORT_SYMBOL
```

`iomem_resource` 利用 `PCI mem` 名字和 `IORESOURCE_MEM (0x00000200)` 标记定义了 `io` 内存的根地址范围。就像上文提到的，我们目前的目的是设置 `iomem` 的结束地址，我们需要这样做：

```
iomem_resource.end = (1ULL << boot_cpu_data.x86_phys_bits) - 1;
```

我们对1左移 `boot_cpu_data.x86_phys_bits` 。 `boot_cpu_data` 是我们在执行 `early_cpu_init` 的时候初始化的 `cpuinfo_x86` 结构。从字面理解， `x86_phys_bits` 代表系统可达到的最大内存地址时需要的比特数。 另外， `iomem_resource` 是通过 `EXPORT_SYMBOL` 宏传递的。这个宏可以把指定的符号（例如 `iomem_resource` ）做动态链接。换句话说，它可以支持动态加载模块的时候访问对应符号。 设置完根 `iomem` 的资源地址范围的结束地址后，下一步就是设置内存映射。它通过调用 `setup_memory_map` 函数实现：

```
void __init setup_memory_map(void)
{
        char *who;

        who = x86_init.resources.memory_setup();
        memcpy(&e820_saved, &e820, sizeof(struct e820map));
        printk(KERN_INFO "e820: BIOS-provided physical RAM map:\n");
        e820_print_map(who);
}
```

首先，我们来看下 `x86_init.resources.memory_setup` 。 `x86_init` 是一种 `x86_init_ops` 类型的结构体，用来表示项资源初始化， `pci` 初始化平台特定的一些设置函数。 `x86_init` 的初始化实现在 arch/x86/kernel/x86_init.c 文件中。我不会全部解释这个初始化过程，因为我们只关心一个地方：

```
struct x86_init_ops x86_init __initdata = {
    .resources = {
            .probe_roms             = probe_roms,
            .reserve_resources      = reserve_standard_io_resources,
            .memory_setup           = default_machine_specific_memory_setup,
    },
    ...
    ...
    ...
}
```

我们可以看到，这里的 `memory_setup` 赋值为 `default_machine_specific_memory_setup`，它是我们在对 内核启动 过程中的所有 e820 条目经过整理和把内存分区填入 `e820map` 结构体中获得的。所有收集的内存分区会用 `printk` 打印出来。你可以通过运行 `dmesg` 命令找到类似于下面的信息：

```
[    0.000000] e820: BIOS-provided physical RAM map:
[    0.000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009d7ff] usable
[    0.000000] BIOS-e820: [mem 0x000000000009d800-0x000000000009ffff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000000e0000-0x00000000000fffff] reserved
[    0.000000] BIOS-e820: [mem 0x0000000000100000-0x00000000be825fff] usable
[    0.000000] BIOS-e820: [mem 0x00000000be826000-0x00000000be82cfff] ACPI NVS
[    0.000000] BIOS-e820: [mem 0x00000000be82d000-0x00000000bf744fff] usable
[    0.000000] BIOS-e820: [mem 0x00000000bf745000-0x00000000bfff4fff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000bfff5000-0x00000000dc041fff] usable
[    0.000000] BIOS-e820: [mem 0x00000000dc042000-0x00000000dc0d2fff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000dc0d3000-0x00000000dc138fff] usable
[    0.000000] BIOS-e820: [mem 0x00000000dc139000-0x00000000dc27dfff] ACPI NVS
[    0.000000] BIOS-e820: [mem 0x00000000dc27e000-0x00000000deffefff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000defff000-0x00000000deffffff] usable
...
...
...
```

复制 `BIOS` 增强磁盘设备信息

下面两部是通过 `parse_setup_data` 函数解析 `setup_data`，并且把 `BIOS` 的 `EDD` 信息复制到安全的地方。`setup_data` 是内核启动头中包含的字段，我们可以在 `x86` 的启动协议中了解：

```
Field name:     setup_data
Type:          write (special)
Offset/size:     0x250/8
Protocol:      2.09+

  The 64-bit physical pointer to NULL terminated single linked list of
  struct setup_data. This is used to define a more extensible boot
  parameters passing mechanism.
```

它用来存储不同类型的设置信息，例如设备树 `blob` ， `EFI` 设置数据等等。第二步是从 `boot_params` 结构中复制我们在 arch/x86/boot/edd.c 中 `BIOS` 的 `EDD` 信息到 `edd` 结构中。

```
static inline void __init copy_edd(void)
{
     memcpy(edd.mbr_signature, boot_params.edd_mbr_sig_buffer,
            sizeof(edd.mbr_signature));
     memcpy(edd.edd_info, boot_params.eddbuf, sizeof(edd.edd_info));
     edd.mbr_signature_nr = boot_params.edd_mbr_sig_buf_entries;
     edd.edd_info_nr = boot_params.eddbuf_entries;
}
```

内存描述符初始化

下一步是在初始化阶段完成内存描述符的初始化。我们知道每个进程都有自己的运行内存地址空间。通过调用 `memory descriptor` 可以看到这些特殊数据结构。 在 linux 内核源码中内存描述符是用 `mm_struct` 结构体表示的。 `mm_struct` 包含许多不同的与进程地址空间有关的字段，像内核代码/数据段的起始和结束地址， `brk` 的起始和结束，内存区域的数量，内存区域列表等。这些结构定义在 include/linux/mm_types.h 中。 `task_struct` 结构的 `mm` 和 `active_mm` 字段包含了每个进程自己的内存描述符。 我们的第一个 `init` 进程也有自己的内存描述符。在之前的章节我们看到过通过 `INIT_TASK` 宏实现 `task_struct` 的部分初始化信息：

```
#define INIT_TASK(tsk)  \
{
    ...
    ...
    ...
    .mm = NULL,         \
    .active_mm  = &init_mm, \
    ...
}
```

`mm` 指向进程地址空间， `active_mm` 指向像内核线程这样子不存在地址空间的有效地址空间（你可以在这个文档 中了解更多内容）。 接下来我们在初始化阶段完成内存描述符中内核代码段，数据段和 `brk` 段的初始化：

```
    init_mm.start_code = (unsigned long) _text;
    init_mm.end_code = (unsigned long) _etext;
    init_mm.end_data = (unsigned long) _edata;
    init_mm.brk = _brk_end;
```

`init_mm` 是初始化阶段的内存描述符定义：

```
struct mm_struct init_mm = {
    .mm_rb          = RB_ROOT,
    .pgd            = swapper_pg_dir,
    .mm_users       = ATOMIC_INIT(2),
    .mm_count       = ATOMIC_INIT(1),
    .mmap_sem       = __RWSEM_INITIALIZER(init_mm.mmap_sem),
    .page_table_lock =  __SPIN_LOCK_UNLOCKED(init_mm.page_table_lock),
    .mmlist         = LIST_HEAD_INIT(init_mm.mmlist),
    INIT_MM_CONTEXT(init_mm)
};
```

其中 `mm_rb` 是虚拟内存区域的红黑树结构， `pgd` 是全局页目录的指针， `mm_user` 是使用该内存空间的进程数目， `mm_count` 是主引用计数， `mmap_sem` 是内存区域信号量。 在初始化阶段完成内存描述符的设置后，下一步是通过 `mpx_mm_init` 完成 `Intel` 内存保护扩展的初始化。下一步是代码/数据/ `bss` 资源的初始化：

```
code_resource.start = __pa_symbol(_text);
    code_resource.end = __pa_symbol(_etext)-1;
    data_resource.start = __pa_symbol(_etext);
    data_resource.end = __pa_symbol(_edata)-1;
    bss_resource.start = __pa_symbol(__bss_start);
    bss_resource.end = __pa_symbol(__bss_stop)-1;
```

通过上面我们已经知道了一小部分关于 `resource` 结构体的样子。在这里，我们把物理地址段赋值给代码/数据/ `bss` 段。你可以在 `/proc/iomem` 中看到：

```
00100000-be825fff : System RAM
  01000000-015bb392 : Kernel code
  015bb393-01930c3f : Kernel data
  01a11000-01ac3fff : Kernel bss
在 [arch/x86/kernel/setup.c](https://github.com/torvalds/linux/blob/16f73eb02d7e1765cca
b3d2018e0bd98eb93d973/arch/x86/kernel/setup.c) 中有所有这些结构体的定义：
static struct resource code_resource = {
    .name    = "Kernel code",
    .start    = 0,
    .end    = 0,
    .flags    = IORESOURCE_BUSY | IORESOURCE_MEM
};
```

本章节涉及的最后一部分就是 `NX` 配置。`NX-bit` 或者 `no-execute` 位是页目录条目的第 63 比特位。它的作用是控制被映射的物理页面是否具有执行代码的能力。 这个比特位只会在通过把 `EFER.NXE` 置为1使能 `no-execute` 页保护机制的时候被使用/设置。在 `x86_configure_nx` 函数中会检查 `CPU` 是否支持 `NX-bit`，以及是否被禁用。 经过检查后，我们会根据结果给 `_supported_pte_mask` 赋值：

```
void x86_configure_nx(void)
{
        if (cpu_has_nx && !disable_nx)
                __supported_pte_mask |= _PAGE_NX;
        else
                __supported_pte_mask &= ~_PAGE_NX;
}
```

结论

以上是 linux 内核初始化过程的第五部分。在这一章我们讲解了有关架构初始化的 `setup_arch` 函数。内容很多，但是我们还没有学习完。其中，`setup_arch` 是一个很复杂的函数，甚至我不确定我们能在以后的章节中讲完它的所有内容。在这一章节中有一些很有趣的概念像 `Fix-mapped` 地址，`ioremap` 等等。 如果没听明白也不用担心，在 内核内存管理，第二部分 还会有更详细的解释。在下一章节我们会继续讲解有关结构初始化的东西， 以及初期内核参数的解析，`pci` 设备的早期转存，直接媒体接口扫描等等。

如果你有任何问题或者建议，你可以留言，也可以直接发送消息给我twitter。

很抱歉，英语并不是我的母语，非常抱歉给您阅读带来不便，如果你发现文中描述有任何问题，清提交一个 **PR** 到 **linux-insides**。

链接

- mm vs active_mm
- e820
- Supervisor mode access prevention
- Kernel stacks
- TSS
- IDT
- Memory mapped I/O
- CFI directives
- PDF. dwarf4 specification
- Call stack
- 内核初始化. Part 4.

内核初始化 第六部分

仍旧是与系统架构有关的初始化

在之前的章节我们从 arch/x86/kernel/setup.c了解了特定于系统架构的初始化事务(在我们的例子中是 `x86_64` 架构)，并且通过 `x86_configure_nx` 函数根据对NX bit的支持配置了 `_PAGE_NX` 标志位。正如我之前写的, `setup_arch` 函数和 `start_kernel` 都非常复杂，所以在这个和下个章节我们将继续学习关于系统架构初始化进程的内容。 `x86_configure_nx` 函数的下面是 `parse_early_param` 函数。这个函数定义在 init/main.c 中并且你可以从它的名字中了解到，这个函数解析内核命令行并且基于给定的参数创建不同的服务 (所有的内核命令行参数你都可以在 Documentation/kernel-parameters.txt 找到)。 你可能记得在最前面的 章节 我们是怎样创建 `earlyprintk` 地。在前面我们用 arch/x86/boot/cmdline.c 里面的 `cmdline_find_option` 和 `__cmdline_find_option`, `__cmdline_find_option_bool` 函数的帮助下寻找内核参数及其值。我们在通用内核部分不依赖于特定的系统架构，在这里我们使用另一种方法。 如果你正在阅读linux内核源代码，你可能注意到这样的调用：

```
early_param("gbpages", parse_direct_gbpages_on);
```

`early_param` 宏需要两个参数:

- 命令行参数的名称
- 如果给定的参数通过，函数将被调用

函数定义如下:

```
#define early_param(str, fn) \
        __setup_param(str, fn, fn, 1)
```

这个定义可以在 include/linux/init.h 中可以找到.
正如你所看到的， `early_param` 宏只是调用了 `__setup_param` 宏:

```
#define __setup_param(str, unique_id, fn, early)                \
        static const char __setup_str_##unique_id[] __initconst \
                __aligned(1) = str; \
        static struct obs_kernel_param __setup_##unique_id      \
                __used __section(.init.setup)                   \
                __attribute__((aligned((sizeof(long)))))        \
                = { __setup_str_##unique_id, fn, early }
```

这个宏内部定义了 `__setup_str_*_id` 变量 (这里的 `*` 取决于给定的函数名称)，然后把给定的命令行参数赋值给这个变量。在下一行中，我们可以看到定义了一个 `obs_kernel_param` 类型的变量 `__setup_ *` 并对其进行初始化。

`obs_kernel_param` 结构体定义如下:

```
struct obs_kernel_param {
        const char *str;
        int (*setup_func)(char *);
        int early;
};
```

这个结构体包含三个字段:

- 内核参数的名称
- 根据不同的参数，选取对应的处理函数
- 决定参数是否为 early 的标记位

注意 `__set_param` 宏定义有 `__section(.init.setup)` 属性。这意味着所有 `__setup_str_ *` 都将被放置在 `.init.setup` 区段中，此外正如我们在 include/asm-generic/vmlinux.lds.h 中看到的，`.init.setup` 区段被放置在 `__setup_start` 和 `__setup_end` 之间:

```
#define INIT_SETUP(initsetup_align)                \
                . = ALIGN(initsetup_align);        \
                VMLINUX_SYMBOL(__setup_start) = .; \
                *(.init.setup)                     \
                VMLINUX_SYMBOL(__setup_end) = .;
```

现在我们知道了参数是怎样定义的，让我们一起回到 `parse_early_param` 的实现上来:

```
void __init parse_early_param(void)
{
        static int done __initdata;
        static char tmp_cmdline[COMMAND_LINE_SIZE] __initdata;

        if (done)
                return;

        /* All fall through to do_early_param. */
        strlcpy(tmp_cmdline, boot_command_line, COMMAND_LINE_SIZE);
        parse_early_options(tmp_cmdline);
        done = 1;
}
```

`parse_early_param` 函数内部定义了两个静态变量。首先第一个变量 `done` 用来检查 `parse_early_param` 函数是否已经被调用过，第二个变量是用来临时存储内核命令行的。然后我们把 `boot_command_line` 的值赋值给刚刚定义的临时命令行变量中(`tmp_cmdline`) 并且从相同的源代码文件 `main.c` 中调用 `parse_early_options` 函数。 `parse_early_options` 函数从 kernel/params.c 中调用 `parse_args` 函数, `parse_args` 解析传入的命令行然后调用 `do_early_param` 函数。 do_early_param 函数 从 `__setup_start` 循环到 `__setup_end` ，如果循环中 `obs_kernel_param` 实例中的 `early` 字段值为1 ,就调用 `obs_kernel_param` 中的第二个函数 `setup_func` 。在这之后所有基于早期命令行参数的服务都已经被创建，在 `parse_early_param` 之后的下一个函数调用是 `x86_report_nx` 。 正如我在这章开头所写的，我们已经用 `x86_configure_nx` 函数配置了 `NX-bit` 位。接下来我们使用 arch/x86/mm/setup_nx.c 中的 `x86_report_nx` 函数打印出关于 `NX` 的信息。注意 `x86_report_nx` 函数不一定在 `x86_configure_nx` 函数之后调用，但是一定在 `parse_early_param` 之后调用。答案很简单: 因为内核支持 `noexec` 参数，所以我们一定在 `parse_early_param` 调用并且解析 `noexec` 参数之后才能调用 `x86_report_nx` :

```
noexec          [X86]
            On X86-32 available only on PAE configured kernels.
            //在X86-32架构上，仅在配置PAE的内核上可用。
            noexec=on: enable non-executable mappings (default)
            //noexec=on:开启非可执行文件的映射(默认)
            noexec=off: disable non-executable mappings
            //noexec=off: 禁用非可执行文件的映射
```

我们可以在启动的时候看到:

```
bootconsole [earlyser0] enabled
NX (Execute Disable) protection: active
SMBIOS 2.8 present.
```

之后我们可以看到下面函数的调用:

```
    memblock_x86_reserve_range_setup_data();
```

这个函数的定义也在 arch/x86/kernel/setup.c 中，然后这个函数为 `setup_data` 重新映射内存并保留内存块(你可以阅读之前的 章节 了解关于 `setup_data` 的更多内容，你也可以在 Linux kernel memory management 中阅读到关于 `ioremap` and `memblock` 的更多内容)。

接下来我们来看看下面的条件语句:

```
    if (acpi_mps_check()) {
#ifdef CONFIG_X86_LOCAL_APIC
        disable_apic = 1;
#endif
        setup_clear_cpu_cap(X86_FEATURE_APIC);
    }
```

`acpi_mps_check` 函数来自于 arch/x86/kernel/acpi/boot.c ，它的结果取决于 `CONFIG_X86_LOCAL_APIC` 和 `CONFIG_x86_MPPARSE` 配置选项:

```
int __init acpi_mps_check(void)
{
#if defined(CONFIG_X86_LOCAL_APIC) && !defined(CONFIG_X86_MPPARSE)
        /* mptable code is not built-in*/
        if (acpi_disabled || acpi_noirq) {
                printk(KERN_WARNING "MPS support code is not built-in.\n"
                       "Using acpi=off or acpi=noirq or pci=noacpi "
                       "may have problem\n");
                 return 1;
        }
#endif
        return 0;
}
```

`acpi_mps_check` 函数检查内置的 `MPS` 又称 多重处理器规范) 表。如果设置了 `CONFIG_X86_LOCAL_APIC` 但未设置 `CONFIG_x86_MPPAARSE` ，而且传递给内核的命令行选项中有 `acpi=off` 、 `acpi=noirq` 或者 `pci=noacpi` 参数，那么 `acpi_mps_check` 函数就会输出警告信息。如果 `acpi_mps_check` 返回了1，这就表示我们禁用了本地 APIC ,而且 `setup_clear_cpu_cap` 宏清除了当前CPU中的 `X86_FEATURE_APIC` 位。（你可以阅读 CPU masks 了解关于CPU mask的更多内容)。

早期的PCI转储

接下来我们通过下面的代码来转储 PCI 设备:

```
#ifdef CONFIG_PCI
    if (pci_early_dump_regs)
        early_dump_pci_devices();
#endif
```

变量 `pci_early_dump_regs` 定义在 arch/x86/pci/common.c 中，他的值取决于内核命令行参数： `pci=earlydump` 。我们可以在drivers/pci/pci.c 中看到这个参数的定义:

```
early_param("pci", pci_setup);
```

`pci_setup` 函数取出 `pci=` 之后的字符串，然后进行解析。这个函数调用 drivers/pci/pci.c 中用 `_weak` 修饰符定义的 `pcibios_setup` 函数，并且每种架构都重写了 `_weak` 修饰过的函数。 例如， `x86_64` 架构上的该函数版本在 arch/x86/pci/common.c 中:

```
char *__init pcibios_setup(char *str) {
        ...
        ...
        ...
        } else if (!strcmp(str, "earlydump")) {
                pci_early_dump_regs = 1;
                return NULL;
        }
        ...
        ...
        ...
}
```

如果我们设置了 `CONFIG_PCI` 选项，而且向内核命令行传递了 `pci=earlydump` 选项，那么 arch/x86/pci/early.c 中的 `early_dump_pci_devices` 函数将会被调用。这个函数像下面这样来检查pci参数 `noearly` ：

```
if (!early_pci_allowed())
        return;
```

如果条件不成立则返回。每个PCI域可以承载多达 `256` 条总线，并且每条总线可以承载多达32个设备。那么接下来我们进入下面的循环:

```
for (bus = 0; bus < 256; bus++) {
                for (slot = 0; slot < 32; slot++) {
                        for (func = 0; func < 8; func++) {
                        ...
                        ...
                        ...
                        }
                }
}
```

然后我们通过 `read_pci_config` 函数来读取 `pci` 配置。

这就是 pci 加载的全部过程了。我们在这里不会深入研究 `pci` 的细节，不过我们会在 `Drivers/PCI` 章节看到更多的细节。

内存解析的完成

在 `early_dump_pci_devices` 函数后面，有一些与可用内存和e820相关的函数，其中 e820 的相关信息我们在 内核安装的第一步 章节中整理过。

```
    /* update the e820_saved too */
    e820_reserve_setup_data();
    finish_e820_parsing();
    ...
    ...
    ...
    e820_add_kernel_range();
    trim_bios_range(void);
    max_pfn = e820_end_of_ram_pfn();
    early_reserve_e820_mpc_new();
```

让我们来一起看看上面的代码。正如你所看到的，第一个函数是 `e820_reserve_setup_data` 。这个函数和我们前面看到的 `memblock_x86_reserve_range_setup_data` 函数做的事情几乎是相同的，但是这个函数同时还会调用 `e820_update_range` 函数，向 `e820map` 中用给定的类型添加新的区域，在我们的例子中，使用的是 `E820_RESERVED_KERN` 类型。接下来的函数是 `finish_e820_parsing`,这个函数使用 `sanitize_e820_map` 函数对 `e820map` 进行清理。除了这两个函数之外，我们还可以看到一些与 e820 有关的函数。你可以在上面的列表中看到这些函数。 `e820_add_kernel_range` 函数需要内核开始和结束的物理地址:

```
u64 start = __pa_symbol(_text);
u64 size = __pa_symbol(_end) - start;
```

函数会检查在 `e820map` 中被标记成 `E820RAM` 的 `.text` `.data` 和 `.bss` 区段，如果没有这些区段，那么就会输出错误信息。接下来的 `trm_bios_range` 函数把 `e820Map` 中的前4096个字节修改为 `E820_RESERVED` 并且再次调用函数 `sanitize_e820_map` 清理 `e820map` 。在这之后我们使用 `e820_end_of_ram_pfn` 函数得到最后一个页帧的编号，每个内存页面都有一个唯一的编号 - `页帧号` ， `e820_end_of_ram_pfn` 函数调用 `e820_end_pfn` 函数返回最大的页面帧号:

```
unsigned long __init e820_end_of_ram_pfn(void)
{
    return e820_end_pfn(MAX_ARCH_PFN);
}
```

`e820_end_pfn` 函数读取特定于系统架构的最大页帧号(对于 `x86_64` 架构来说 `MAX_ARCH_PFN` 是 `0x400000000`)。在 `e820_end_pfn` 函数中我们遍历整个 `e820` 槽，并且检查 `e820` 中是否有 `E820_RAM` 或者 `E820_PRAM` 类型条目，因为我们只能对这些类型计算页面帧号，然后我们得到当前 `e820` 页面帧的基地址和结束地址，同时对这些地址进行检查:

```
for (i = 0; i < e820.nr_map; i++) {
        struct e820entry *ei = &e820.map[i];
        unsigned long start_pfn;
        unsigned long end_pfn;

        if (ei->type != E820_RAM && ei->type != E820_PRAM)
            continue;

        start_pfn = ei->addr >> PAGE_SHIFT;
        end_pfn = (ei->addr + ei->size) >> PAGE_SHIFT;

        if (start_pfn >= limit_pfn)
            continue;
        if (end_pfn > limit_pfn) {
            last_pfn = limit_pfn;
            break;
        }
        if (end_pfn > last_pfn)
            last_pfn = end_pfn;
}
```

```
    if (last_pfn > max_arch_pfn)
        last_pfn = max_arch_pfn;

    printk(KERN_INFO "e820: last_pfn = %#lx max_arch_pfn = %#lx\n",
             last_pfn, max_arch_pfn);
    return last_pfn;
```

接下来我们检查在循环中得到的 `last_pfn` ， `last_pfn` 不得大于特定于系统架构的最大页帧号(在我们的例子中是 `x86_64` 系统架构),然后输出关于最大页帧号的信息，并且返回 `last_pfn` 。我们可以在 `dmesg` 的输出中看到 `last_pfn` ：

```
...
[    0.000000] e820: last_pfn = 0x41f000 max_arch_pfn = 0x400000000
...
```

在这之后，我们计算出了最大的页帧号，我们要计算 `max_low_pfn` ,这是 `低端内存` 或者低于第一个4GB中的最大页面帧。如果系统安装了超过4GB的内存RAM， `max_low_pfn` 将会是 `e820_end_of_low_ram_pfn` 函数的结果，这个函数和 `e820_end_of_ram_pfn` 相似，但是有4GB限制，换句话说 `max_low_pfn` 和 `max_pfn` 的值是一样的:

```
if (max_pfn > (1UL<<(32 - PAGE_SHIFT)))
    max_low_pfn = e820_end_of_low_ram_pfn();
else
    max_low_pfn = max_pfn;

high_memory = (void *)__va(max_pfn * PAGE_SIZE - 1) + 1;
```

接下来我们通过 `__va` 宏计算 `高端内存` (有更高的内存直接映射上界)中的最大页帧号,并且这个宏会根据给定的物理内存返回一个虚拟地址。

桌面管理接口

在处理完不同内存区域和 `e820` 槽之后，接下来就该收集计算机的相关信息了。我们将用下面的函数收集与 桌面管理接口 有关的所有信息:

```
dmi_scan_machine();
dmi_memdev_walk();
```

首先是定义在 drivers/firmware/dmi_scan.c 中的 `dmi_scan_machine` 函数。这个函数遍历 System Management BIOS 结构，并从中提取信息。这里有两种方法来访问 `SMBIOS` 表: 第一种是从 EFI 的配置表获得指向 `SMBIOS` 表的指针；第二种是扫描 `0xF0000` 和 `0x10000` 地址之间的物理内存。让我们一起看看第二种方法。 `dmi_scan_machine` 函数通过 `dmi_early_remap` 函数将 `0xf0000` 和 `0x10000` 之间的内存重新映射并追加到 `early_ioremap` 上:

```
void __init dmi_scan_machine(void)
{
    char __iomem *p, *q;
    char buf[32];
    ...
    ...
    ...
    p = dmi_early_remap(0xF0000, 0x10000);
    if (p == NULL)
            goto error;
```

然后迭代所有的 `DMI` 头部地址，并且查找 `_SM_` 字符串:

```
memset(buf, 0, 16);
for (q = p; q < p + 0x10000; q += 16) {
        memcpy_fromio(buf + 16, q, 16);
        if (!dmi_smbios3_present(buf) || !dmi_present(buf)) {
            dmi_available = 1;
            dmi_early_unmap(p, 0x10000);
            goto out;
        }
        memcpy(buf, buf + 16, 16);
}
```

`_SM_` 字符串一定在 `000F0000h` 和 `0x000FFFFF` 地址之间。在这里我们用 `memcpy_fromio` 函数向 `buf` 里面拷贝16个字节，这个函数和 `memcpy` 函数的作用是一样的。然后对这个缓冲区(`buf`)执行`dmi_smbios3_present` 和 `dmi_present` 函数。这些函数检查 `buf` 的前4个字节是否是 `__SM__` 字符串，并且获得 `SMBIOS` 的版本和 `_DMI_` 的属性例如 `_DMI_` 的结构表长度、结构表的地址等... 在其中的一个函数完成之后，你就可以在 `dmesg` 的输出中看到它的运行结果:

```
[    0.000000] SMBIOS 2.7 present.
[    0.000000] DMI: Gigabyte Technology Co., Ltd. Z97X-UD5H-BK/Z97X-UD5H-BK, BIOS F6 0
6/17/2014
```

在 `dmi_scan_machine` 函数的最后，我们取消之前映射的内存:

```
dmi_early_unmap(p, 0x10000);
```

第二个函数是 - `dmi_memdev_walk` 。和你想的一样，这个函数遍历整个内存设备。让我们一起看看这个函数:

```
void __init dmi_memdev_walk(void)
{
    if (!dmi_available)
        return;

    if (dmi_walk_early(count_mem_devices) == 0 && dmi_memdev_nr) {
        dmi_memdev = dmi_alloc(sizeof(*dmi_memdev) * dmi_memdev_nr);
        if (dmi_memdev)
            dmi_walk_early(save_mem_devices);
    }
}
```

这个函数检查 `DMI` 是否可用(我们之前在 `dmi_scan_machine` 函数中得到了这个结果，并且保存在 `dmi_available` 变量中)，然后使用 `dmi_walk_early` 和 `dmi_alloc` 函数收集内存设备的有关信息,其中 `dmi_alloc` 的定义如下:

```
#ifdef CONFIG_DMI
RESERVE_BRK(dmi_alloc, 65536);
#endif
```

定义在 arch/x86/include/asm/setup.h 中的 `RESERVE_BRK` 函数会在 `brk` 段中预留给定大小的空间:

```
init_hypervisor_platform();
x86_init.resources.probe_roms();
insert_resource(&iomem_resource, &code_resource);
insert_resource(&iomem_resource, &data_resource);
insert_resource(&iomem_resource, &bss_resource);
early_gart_iommu_check();
```

均衡多处理(SMP)的配置

接下来的一步是解析 SMP 的配置信息。我们调用 `find_smp_config` 函数来完成这个任务，这个函数内部调用另一个函数:

```
static inline void find_smp_config(void)
{
        x86_init.mpparse.find_smp_config();
}
```

在函数的内部，`x86_init.mpparse.find_smp_config` 函数就是 arch/x86/kernel/mpparse.c 中的 `default_find_smp_config` 函数。我们调用 `default_find_smp_config` 函数扫描内存中的一些区域来寻找 `SMP` 的配置信息,并在找到它们的时候返回:

```
if (smp_scan_config(0x0, 0x400) ||
            smp_scan_config(639 * 0x400, 0x400) ||
            smp_scan_config(0xF0000, 0x10000))
            return;
```

首先 `smp_scan_config` 函数内部定义了一些变量:

```
unsigned int *bp = phys_to_virt(base);
struct mpf_intel *mpf;
```

第一个变量是我们用来扫描 `SMP` 配置的内存区域的虚拟地址；第二个变量是指向 `mpf_intel` 结构体的指针。让我们一起试着去理解 `mpf_intel` 是什么吧。所有的信息都存储在多处理器配置数据结构中。 `mpf_intel` 就是这个结构，看下来像是下面这样:

```
struct mpf_intel {
        char signature[4];
        unsigned int physptr;
        unsigned char length;
        unsigned char specification;
        unsigned char checksum;
        unsigned char feature1;
        unsigned char feature2;
        unsigned char feature3;
        unsigned char feature4;
        unsigned char feature5;
};
```

正如我们在文档中看到的那样 - 系统 BIOS的主要功能之一就是创建MP浮点型指针结构和MP配置表。而且操作系统必须可以访问关于多处理器配置的有关信息， `mpf_intel` 中存储了多处理器配置表的物理地址(看结构体的第二个变量),然后， `smp_scan_config` 函数在指定的内存区域中循环查找 `MP floating pointer structure` 。这个函数还会检查当前字节是否指向 `SMP` 签名，然后检查签名的校验和，并且检查循环中的 `mpf->specification` 的值是1还是4(这个值只能是1或者是4):

```
while (length > 0) {
if ((*bp == SMP_MAGIC_IDENT) &&
    (mpf->length == 1) &&
    !mpf_checksum((unsigned char *)bp, 16) &&
    ((mpf->specification == 1)
    || (mpf->specification == 4))) {

        mem = virt_to_phys(mpf);
        memblock_reserve(mem, sizeof(*mpf));
        if (mpf->physptr)
            smp_reserve_memory(mpf);
    }
}
```

如果搜索成功，就调用 `memblock_reserve` 函数保留一定的内存块，并且为多处理器配置表保留物理地址。你可以在 MultiProcessor Specification 中找到相关的文档。你也可以在 `SMP` 的特定章节阅读更多细节。

其他的早期内存初始化程序

在 `setup_arch` 的下一步，我们可以看到 `early_alloc_pgt_buf` 函数的调用,这个函数在早期阶段分配页表缓冲区。页表缓冲区将被放置在 `brk` 区段中。让我们一起看看这个功能的实现:

```
void  __init early_alloc_pgt_buf(void)
{
        unsigned long tables = INIT_PGT_BUF_SIZE;
        phys_addr_t base;

        base = __pa(extend_brk(tables, PAGE_SIZE));

        pgt_buf_start = base >> PAGE_SHIFT;
        pgt_buf_end = pgt_buf_start;
        pgt_buf_top = pgt_buf_start + (tables >> PAGE_SHIFT);
}
```

首先这个函数获得页表缓冲区的大小，它的值是 `INIT_PGT_BUF_SIZE` ，这个值在目前的linux 4.0 内核中是 `(6 * PAGE_SIZE)` 。因为我们已经得到了页表缓冲区的大小，现在我们调用 `extend_brk` 函数并且传入两个参数: size和align。你可以从他们的名称中猜到,这个函数扩展 `brk` 区段。正如我们在linux内核链接脚本中看到的， `brk` 区段在内存中的位置恰好就在 BSS 区段后面:

```
    . = ALIGN(PAGE_SIZE);
    .brk : AT(ADDR(.brk) - LOAD_OFFSET) {
        __brk_base = .;
        . += 64 * 1024;         /* 64k alignment slop space */
        *(.brk_reservation)     /* areas brk users have reserved */
        __brk_limit = .;
    }
```

我们也可以使用 `readelf` 工具来找到它:

```
  [25] .bss              NOBITS           ffffffff8199d000  00d9d000
       00000000000b4000  0000000000000000  WA       0     0     4096
  [26] .brk              NOBITS           ffffffff81a51000  00d9d000
       0000000000026000  0000000000000000  WA       0     0     1
```

之后我们用 `_pa` 宏得到了新的 `brk` 区段的物理地址，我们计算页表缓冲区的基地址和结束地址。因为我们之前已经创建好了页面缓冲区，所以现在我们使用 `reserve_brk` 函数为 `brk` 区段保留内存块:

```
static void __init reserve_brk(void)
{
    if (_brk_end > _brk_start)
        memblock_reserve(__pa_symbol(_brk_start),
                 _brk_end - _brk_start);

    _brk_start = 0;
}
```

注意在 `reserve_brk` 的最后，我们把 `_brk_start` 赋值为0,因为在这之后我们不会再为 `brk` 分配内存了，我们需要使用 `cleanup_highmap` 函数来释放内核映射中越界的内存区域。请记住内核映射是 `__START_KERNEL_map` 和 `_end - _text` 或者 `level2_kernel_pgt` 对内核 `_text` 、 `data` 和 `bss` 区段的映射。在 `clean_high_map` 的开始部分我们定义下面这些参数:

```
unsigned long vaddr = __START_KERNEL_map;
unsigned long end = roundup((unsigned long)_end, PMD_SIZE) - 1;
pmd_t *pmd = level2_kernel_pgt;
pmd_t *last_pmd = pmd + PTRS_PER_PMD;
```

现在，因为我们已经定义了内核映射的开始和结束位置，所以我们在循环中遍历所有内核页中间目录条目, 并且清除不在 `_text` 和 `end` 区段中的条目:

```
for (; pmd < last_pmd; pmd++, vaddr += PMD_SIZE) {
        if (pmd_none(*pmd))
            continue;
        if (vaddr < (unsigned long) _text || vaddr > end)
            set_pmd(pmd, __pmd(0));
}
```

在这之后，我们使用 `memblock_set_current_limit` (你可以在linux 内存管理 第二章节 阅读关于 `memblock` 的更多内容) 函数来为 `memblock` 分配内存设置一个界限，这个界限可以是 `ISA_END_ADDRESS` 或者 `0x100000` ，然后调用 `memblock_x86_fill` 函数根据 `e820` 来填充 `memblock` 相关信息。你可以在内核初始化的时候看到这个函数运行的结果:

```
MEMBLOCK configuration:
 memory size = 0x1fff7ec00 reserved size = 0x1e30000
 memory.cnt  = 0x3
 memory[0x0]     [0x00000000001000-0x0000000009efff], 0x9e000 bytes flags: 0x0
 memory[0x1]     [0x00000000100000-0x000000bffdffff], 0xbfee0000 bytes flags: 0x0
 memory[0x2]     [0x00000100000000-0x0000023fffffff], 0x140000000 bytes flags: 0x0
 reserved.cnt  = 0x3
 reserved[0x0]     [0x0000000009f000-0x000000000fffff], 0x61000 bytes flags: 0x0
 reserved[0x1]     [0x00000001000000-0x00000001a57fff], 0xa58000 bytes flags: 0x0
 reserved[0x2]     [0x0000007ec89000-0x0000007fffffff], 0x1377000 bytes flags: 0x0
```

除了 `memblock_x86_fill` 之外的其他函数还有: `early_reserve_e820_mpc_new` 函数在 `e820map` 中为多处理器规格表分配额外的槽， `reserve_real_mode` - 用于保留从 `0x0` 到1M的低端内存用作到实模式的跳板(用于重启等...)， `trim_platform_memory_ranges` 函数用于清除掉以 `0x20050000` , `0x20110000` 等地址开头的内存空间。这些内存区域必须被排除在外，因为 Sandy Bridge 会在这些内存区域出现一些问题， `trim_low_memory_range` 函数用于保留 `memblock` 中的前4KB页面， `init_mem_mapping` 函数用于在 `PAGE_OFFSET` 处重建物理内存的直接映射, `early_trap_pf_init` 函数用于建立 `#PF` 处理函数(我们将会在有关中断的章节看到它), `setup_real_mode` 函数用于建立到 实模式 代码的跳板。

这就是本章的全部内容了。您可能注意到这部分并没有包括 `setup_arch` 中的所有函数 (如 `early_gart_iommu_check` 、mtrr 的初始化函数等...)。正如我已经说了很多次的, `setup_arch` 函数很复杂，linux内核也很复杂。这就是为什么我不能包括linux内核中的每一行代码。我认为我们并没有错过重要的东西, 但是你可能会说: 每行代码都很重要。是的, 这没错, 但不管怎样我略过了他们, 因为我认为对于整个linux内核面面俱到是不现实的。无论如何, 我们会经常复习所学的内容, 如果有什么不熟悉的内容, 我们将会深入研究这些内容。

结束语

这里是linux 内核初始化进程第六章节的结尾。在这一章节中，我们再次深入研究了 `setup_arch` 函数，然而这是个很长的部分，我们目前还没有学习完。的确, `setup_arch` 很复杂，希望下个章节将会是这个函数的最后一个部分。。

如果你有任何的疑问或者建议，你可以留言，也可以直接发消息给我twitter。

很抱歉，英语并不是我的母语，非常抱歉给您阅读带来不便，如果你发现文中描述有任何问题，请提交一个 **PR** 到 **linux-insides**.

链接

- MultiProcessor Specification
- NX bit
- Documentation/kernel-parameters.txt
- APIC
- CPU masks
- Linux kernel memory management
- PCI
- e820
- System Management BIOS
- System Management BIOS
- EFI

- SMP
- MultiProcessor Specification
- BSS
- SMBIOS specification
- 前一个章节

Kernel initialization. Part 7.

The End of the architecture-specific initialization, almost...

This is the seventh part of the Linux Kernel initialization process which covers insides of the `setup_arch` function from the arch/x86/kernel/setup.c. As you can know from the previous parts, the `setup_arch` function does some architecture-specific (in our case it is x86_64) initialization stuff like reserving memory for kernel code/data/bss, early scanning of the Desktop Management Interface, early dump of the PCI device and many many more. If you have read the previous part, you can remember that we've finished it at the `setup_real_mode` function. In the next step, as we set limit of the memblock to the all mapped pages, we can see the call of the `setup_log_buf` function from the kernel/printk/printk.c.

The `setup_log_buf` function setups kernel cyclic buffer and its length depends on the `CONFIG_LOG_BUF_SHIFT` configuration option. As we can read from the documentation of the `CONFIG_LOG_BUF_SHIFT` it can be between `12` and `21`. In the insides, buffer defined as array of chars:

```
#define __LOG_BUF_LEN (1 << CONFIG_LOG_BUF_SHIFT)
static char __log_buf[__LOG_BUF_LEN] __aligned(LOG_ALIGN);
static char *log_buf = __log_buf;
```

Now let's look on the implementation of the `setup_log_buf` function. It starts with check that current buffer is empty (It must be empty, because we just setup it) and another check that it is early setup. If setup of the kernel log buffer is not early, we call the `log_buf_add_cpu` function which increase size of the buffer for every CPU:

```
if (log_buf != __log_buf)
    return;

if (!early && !new_log_buf_len)
    log_buf_add_cpu();
```

We will not research `log_buf_add_cpu` function, because as you can see in the `setup_arch`, we call `setup_log_buf` as:

```
setup_log_buf(1);
```

where `1` means that it is early setup. In the next step we check `new_log_buf_len` variable which is updated length of the kernel log buffer and allocate new space for the buffer with the `memblock_virt_alloc` function for it, or just return.

As kernel log buffer is ready, the next function is `reserve_initrd`. You can remember that we already called the `early_reserve_initrd` function in the fourth part of the Kernel initialization. Now, as we reconstructed direct memory mapping in the `init_mem_mapping` function, we need to move initrd into directly mapped memory. The `reserve_initrd` function starts from the definition of the base address and end address of the `initrd` and check that `initrd` is provided by a bootloader. All the same as what we saw in the `early_reserve_initrd`. But instead of the reserving place in the `memblock` area with the call of the `memblock_reserve` function, we get the mapped size of the direct memory area and check that the size of the `initrd` is not greater than this area with:

```
mapped_size = memblock_mem_size(max_pfn_mapped);
if (ramdisk_size >= (mapped_size>>1))
    panic("initrd too large to handle, "
          "disabling initrd (%lld needed, %lld available)\n",
          ramdisk_size, mapped_size>>1);
```

You can see here that we call `memblock_mem_size` function and pass the `max_pfn_mapped` to it, where `max_pfn_mapped` contains the highest direct mapped page frame number. If you do not remember what is `page frame number`, explanation is simple: First `12` bits of the virtual address represent offset in the physical page or page frame. If we right-shift out `12` bits of the virtual address, we'll discard offset part and will get `Page Frame Number`. In the `memblock_mem_size` we go through the all memblock `mem` (not reserved) regions and calculates size of the mapped pages and return it to the `mapped_size` variable (see code above). As we got amount of the direct mapped memory, we check that size of the `initrd` is not greater than mapped pages. If it is greater we just call `panic` which halts the system and prints famous Kernel panic message. In the next step we print information about the `initrd` size. We can see the result of this in the `dmesg` output:

```
[0.000000] RAMDISK: [mem 0x36d20000-0x37687fff]
```

and relocate `initrd` to the direct mapping area with the `relocate_initrd` function. In the start of the `relocate_initrd` function we try to find a free area with the `memblock_find_in_range` function:

```
relocated_ramdisk = memblock_find_in_range(0, PFN_PHYS(max_pfn_mapped), area_size, PAG
E_SIZE);

if (!relocated_ramdisk)
    panic("Cannot find place for new RAMDISK of size %lld\n",
           ramdisk_size);
```

The `memblock_find_in_range` function tries to find a free area in a given range, in our case from `0` to the maximum mapped physical address and size must equal to the aligned size of the `initrd`. If we didn't find a area with the given size, we call `panic` again. If all is good, we start to relocated RAM disk to the down of the directly mapped memory in the next step.

In the end of the `reserve_initrd` function, we free memblock memory which occupied by the ramdisk with the call of the:

```
memblock_free(ramdisk_image, ramdisk_end - ramdisk_image);
```

After we relocated `initrd` ramdisk image, the next function is `vsmp_init` from the arch/x86/kernel/vsmp_64.c. This function initializes support of the `ScaleMP vSMP`. As I already wrote in the previous parts, this chapter will not cover non-related `x86_64` initialization parts (for example as the current or `ACPI`, etc.). So we will skip implementation of this for now and will back to it in the part which cover techniques of parallel computing.

The next function is `io_delay_init` from the arch/x86/kernel/io_delay.c. This function allows to override default default I/O delay `0x80` port. We already saw I/O delay in the Last preparation before transition into protected mode, now let's look on the `io_delay_init` implementation:

```
void __init io_delay_init(void)
{
    if (!io_delay_override)
        dmi_check_system(io_delay_0xed_port_dmi_table);
}
```

This function check `io_delay_override` variable and overrides I/O delay port if `io_delay_override` is set. We can set `io_delay_override` variably by passing `io_delay` option to the kernel command line. As we can read from the Documentation/kernel-parameters.txt, `io_delay` option is:

```
io_delay=    [X86] I/O delay method
    0x80
        Standard port 0x80 based delay
    0xed
        Alternate port 0xed based delay (needed on some systems)
    udelay
        Simple two microseconds delay
    none
        No delay
```

We can see `io_delay` command line parameter setup with the `early_param` macro in the arch/x86/kernel/io_delay.c

```
early_param("io_delay", io_delay_param);
```

More about `early_param` you can read in the previous part. So the `io_delay_param` function which setups `io_delay_override` variable will be called in the do_early_param function. `io_delay_param` function gets the argument of the `io_delay` kernel command line parameter and sets `io_delay_type` depends on it:

```
static int __init io_delay_param(char *s)
{
        if (!s)
                return -EINVAL;

        if (!strcmp(s, "0x80"))
                io_delay_type = CONFIG_IO_DELAY_TYPE_0X80;
        else if (!strcmp(s, "0xed"))
                io_delay_type = CONFIG_IO_DELAY_TYPE_0XED;
        else if (!strcmp(s, "udelay"))
                io_delay_type = CONFIG_IO_DELAY_TYPE_UDELAY;
        else if (!strcmp(s, "none"))
                io_delay_type = CONFIG_IO_DELAY_TYPE_NONE;
        else
                return -EINVAL;

        io_delay_override = 1;
        return 0;
}
```

The next functions are `acpi_boot_table_init`, `early_acpi_boot_init` and `initmem_init` after the `io_delay_init`, but as I wrote above we will not cover ACPI related stuff in this `Linux Kernel initialization process` chapter.

Allocate area for DMA

In the next step we need to allocate area for the Direct memory access with the `dma_contiguous_reserve` function which is defined in the drivers/base/dma-contiguous.c. `DMA` is a special mode when devices communicate with memory without CPU. Note that we pass one parameter - `max_pfn_mapped << PAGE_SHIFT`, to the `dma_contiguous_reserve` function and as you can understand from this expression, this is limit of the reserved memory. Let's look on the implementation of this function. It starts from the definition of the following variables:

```
phys_addr_t selected_size = 0;
phys_addr_t selected_base = 0;
phys_addr_t selected_limit = limit;
bool fixed = false;
```

where first represents size in bytes of the reserved area, second is base address of the reserved area, third is end address of the reserved area and the last `fixed` parameter shows where to place reserved area. If `fixed` is `1` we just reserve area with the `memblock_reserve`, if it is `0` we allocate space with the `kmemleak_alloc`. In the next step we check `size_cmdline` variable and if it is not equal to `-1` we fill all variables which you can see above with the values from the `cma` kernel command line parameter:

```
if (size_cmdline != -1) {
   ...
   ...
   ...
}
```

You can find in this source code file definition of the early parameter:

```
early_param("cma", early_cma);
```

where `cma` is:

```
cma=nn[MG]@[start[MG][-end[MG]]]
        [ARM,X86,KNL]
        Sets the size of kernel global memory area for
        contiguous memory allocations and optionally the
        placement constraint by the physical address range of
        memory allocations. A value of 0 disables CMA
        altogether. For more information, see
        include/linux/dma-contiguous.h
```

If we will not pass `cma` option to the kernel command line, `size_cmdline` will be equal to `-1`. In this way we need to calculate size of the reserved area which depends on the following kernel configuration options:

- `CONFIG_CMA_SIZE_SEL_MBYTES` - size in megabytes, default global `CMA` area, which is equal to `CMA_SIZE_MBYTES * SZ_1M` or `CONFIG_CMA_SIZE_MBYTES * 1M`;
- `CONFIG_CMA_SIZE_SEL_PERCENTAGE` - percentage of total memory;
- `CONFIG_CMA_SIZE_SEL_MIN` - use lower value;
- `CONFIG_CMA_SIZE_SEL_MAX` - use higher value.

As we calculated the size of the reserved area, we reserve area with the call of the `dma_contiguous_reserve_area` function which first of all calls:

```
ret = cma_declare_contiguous(base, size, limit, 0, 0, fixed, res_cma);
```

function. The `cma_declare_contiguous` reserves contiguous area from the given base address with given size. After we reserved area for the `DMA`, next function is the `memblock_find_dma_reserve`. As you can understand from its name, this function counts the reserved pages in the `DMA` area. This part will not cover all details of the `CMA` and `DMA`, because they are big. We will see much more details in the special part in the Linux Kernel Memory management which covers contiguous memory allocators and areas.

Initialization of the sparse memory

The next step is the call of the function - `x86_init.paging.pagetable_init`. If you try to find this function in the linux kernel source code, in the end of your search, you will see the following macro:

```
#define native_pagetable_init        paging_init
```

which expands as you can see to the call of the `paging_init` function from the arch/x86/mm/init_64.c. The `paging_init` function initializes sparse memory and zone sizes. First of all what's zones and what is it `Sparsemem`. The `Sparsemem` is a special foundation in the linux kernel memory manager which used to split memory area into different memory banks in the NUMA systems. Let's look on the implementation of the `paginig_init` function:

```
void __init paging_init(void)
{
        sparse_memory_present_with_active_regions(MAX_NUMNODES);
        sparse_init();

        node_clear_state(0, N_MEMORY);
        if (N_MEMORY != N_NORMAL_MEMORY)
                node_clear_state(0, N_NORMAL_MEMORY);

        zone_sizes_init();
}
```

As you can see there is call of the `sparse_memory_present_with_active_regions` function which records a memory area for every `NUMA` node to the array of the `mem_section` structure which contains a pointer to the structure of the array of `struct page`. The next `sparse_init` function allocates non-linear `mem_section` and `mem_map`. In the next step we clear state of the movable memory nodes and initialize sizes of zones. Every `NUMA` node is divided into a number of pieces which are called - `zones`. So, `zone_sizes_init` function from the arch/x86/mm/init.c initializes size of zones.

Again, this part and next parts do not cover this theme in full details. There will be special part about `NUMA`.

vsyscall mapping

The next step after `SparseMem` initialization is setting of the `trampoline_cr4_features` which must contain content of the `cr4` Control register. First of all we need to check that current CPU has support of the `cr4` register and if it has, we save its content to the `trampoline_cr4_features` which is storage for `cr4` in the real mode:

```
if (boot_cpu_data.cpuid_level >= 0) {
    mmu_cr4_features = __read_cr4();
    if (trampoline_cr4_features)
        *trampoline_cr4_features = mmu_cr4_features;
}
```

The next function which you can see is `map_vsyscal` from the arch/x86/kernel/vsyscall_64.c. This function maps memory space for vsyscalls and depends on `CONFIG_X86_VSYSCALL_EMULATION` kernel configuration option. Actually `vsyscall` is a special segment which provides fast access to the certain system calls like `getcpu`, etc. Let's look on implementation of this function:

```
void __init map_vsyscall(void)
{
        extern char __vsyscall_page;
        unsigned long physaddr_vsyscall = __pa_symbol(&__vsyscall_page);

        if (vsyscall_mode != NONE)
                __set_fixmap(VSYSCALL_PAGE, physaddr_vsyscall,
                             vsyscall_mode == NATIVE
                             ? PAGE_KERNEL_VSYSCALL
                             : PAGE_KERNEL_VVAR);

        BUILD_BUG_ON((unsigned long)__fix_to_virt(VSYSCALL_PAGE) !=
                     (unsigned long)VSYSCALL_ADDR);
}
```

In the beginning of the `map_vsyscall` we can see definition of two variables. The first is extern variable `__vsyscall_page`. As a extern variable, it defined somewhere in other source code file. Actually we can see definition of the `__vsyscall_page` in the arch/x86/kernel/vsyscall_emu_64.S. The `__vsyscall_page` symbol points to the aligned calls of the `vsyscalls` as `gettimeofday`, etc.:

```
    .globl __vsyscall_page
    .balign PAGE_SIZE, 0xcc
    .type __vsyscall_page, @object
__vsyscall_page:

    mov $__NR_gettimeofday, %rax
    syscall
    ret

    .balign 1024, 0xcc
    mov $__NR_time, %rax
    syscall
    ret
    ...
    ...
    ...
```

The second variable is `physaddr_vsyscall` which just stores physical address of the `__vsyscall_page` symbol. In the next step we check the `vsyscall_mode` variable, and if it is not equal to `NONE`, it is `EMULATE` by default:

```
static enum { EMULATE, NATIVE, NONE } vsyscall_mode = EMULATE;
```

And after this check we can see the call of the `__set_fixmap` function which calls `native_set_fixmap` with the same parameters:

```
void native_set_fixmap(enum fixed_addresses idx, unsigned long phys, pgprot_t flags)
{
        __native_set_fixmap(idx, pfn_pte(phys >> PAGE_SHIFT, flags));
}

void __native_set_fixmap(enum fixed_addresses idx, pte_t pte)
{
        unsigned long address = __fix_to_virt(idx);

        if (idx >= __end_of_fixed_addresses) {
                BUG();
                return;
        }
        set_pte_vaddr(address, pte);
        fixmaps_set++;
}
```

Here we can see that `native_set_fixmap` makes value of `Page Table Entry` from the given physical address (physical address of the `__vsyscall_page` symbol in our case) and calls internal function - `__native_set_fixmap`. Internal function gets the virtual address of the given `fixed_addresses` index (`VSYSCALL_PAGE` in our case) and checks that given index is not greater than end of the fix-mapped addresses. After this we set page table entry with the call of the `set_pte_vaddr` function and increase count of the fix-mapped addresses. And in the end of the `map_vsyscall` we check that virtual address of the `VSYSCALL_PAGE` (which is first index in the `fixed_addresses`) is not greater than `VSYSCALL_ADDR` which is `-10UL << 20` or `ffffffffff600000` with the `BUILD_BUG_ON` macro:

```
BUILD_BUG_ON((unsigned long)__fix_to_virt(VSYSCALL_PAGE) !=
                     (unsigned long)VSYSCALL_ADDR);
```

Now `vsyscall` area is in the `fix-mapped` area. That's all about `map_vsyscall`, if you do not know anything about fix-mapped addresses, you can read Fix-Mapped Addresses and ioremap. We will see more about `vsyscalls` in the `vsyscalls and vdso` part.

Getting the SMP configuration

You may remember how we made a search of the SMP configuration in the previous part. Now we need to get the `SMP` configuration if we found it. For this we check `smp_found_config` variable which we set in the `smp_scan_config` function (read about it the previous part) and call the `get_smp_config` function:

```
if (smp_found_config)
    get_smp_config();
```

The `get_smp_config` expands to the `x86_init.mpparse.default_get_smp_config` function which is defined in the arch/x86/kernel/mpparse.c. This function defines a pointer to the multiprocessor floating pointer structure - `mpf_intel` (you can read about it in the previous part) and does some checks:

```
struct mpf_intel *mpf = mpf_found;

if (!mpf)
    return;

if (acpi_lapic && early)
   return;
```

Here we can see that multiprocessor configuration was found in the `smp_scan_config` function or just return from the function if not. The next check is `acpi_lapic` and `early`. And as we did this checks, we start to read the `SMP` configuration. As we finished reading it, the next step is - `prefill_possible_map` function which makes preliminary filling of the possible CPU's `cpumask` (more about it you can read in the Introduction to the cpumasks).

The rest of the setup_arch

Here we are getting to the end of the `setup_arch` function. The rest of function of course is important, but details about these stuff will not will not be included in this part. We will just take a short look on these functions, because although they are important as I wrote above, but they cover non-generic kernel features related with the `NUMA`, `SMP`, `ACPI` and `APICs`, etc. First of all, the next call of the `init_apic_mappings` function. As we can understand this function sets the address of the local APIC. The next is `x86_io_apic_ops.init` and this function initializes I/O APIC. Please note that we will see all details related with `APIC` in the chapter about interrupts and exceptions handling. In the next step we reserve standard I/O resources like `DMA`, `TIMER`, `FPU`, etc., with the call of the `x86_init.resources.reserve_resources` function. Following is `mcheck_init` function initializes `Machine check Exception` and the last is `register_refined_jiffies` which registers jiffy (There will be separate chapter about timers in the kernel).

So that's all. Finally we have finished with the big `setup_arch` function in this part. Of course as I already wrote many times, we did not see full details about this function, but do not worry about it. We will be back more than once to this function from different chapters for understanding how different platform-dependent parts are initialized.

That's all, and now we can back to the `start_kernel` from the `setup_arch`.

Back to the main.c

As I wrote above, we have finished with the `setup_arch` function and now we can back to the `start_kernel` function from the init/main.c. As you may remember or saw yourself, `start_kernel` function as big as the `setup_arch`. So the couple of the next part will be dedicated to learning of this function. So, let's continue with it. After the `setup_arch` we can see the call of the `mm_init_cpumask` function. This function sets the cpumask pointer to the memory descriptor `cpumask`. We can look on its implementation:

```
static inline void mm_init_cpumask(struct mm_struct *mm)
{
#ifdef CONFIG_CPUMASK_OFFSTACK
        mm->cpu_vm_mask_var = &mm->cpumask_allocation;
#endif
        cpumask_clear(mm->cpu_vm_mask_var);
}
```

As you can see in the init/main.c, we pass memory descriptor of the init process to the `mm_init_cpumask` and depends on `CONFIG_CPUMASK_OFFSTACK` configuration option we clear TLB switch `cpumask`.

In the next step we can see the call of the following function:

```
setup_command_line(command_line);
```

This function takes pointer to the kernel command line allocates a couple of buffers to store command line. We need a couple of buffers, because one buffer used for future reference and accessing to command line and one for parameter parsing. We will allocate space for the following buffers:

- `saved_command_line` - will contain boot command line;
- `initcall_command_line` - will contain boot command line. will be used in the `do_initcall_level`;
- `static_command_line` - will contain command line for parameters parsing.

We will allocate space with the `memblock_virt_alloc` function. This function calls `memblock_virt_alloc_try_nid` which allocates boot memory block with `memblock_reserve` if slab is not available or uses `kzalloc_node` (more about it will be in the linux memory management chapter). The `memblock_virt_alloc` uses `BOOTMEM_LOW_LIMIT` (physical

address of the `(PAGE_OFFSET + 0x1000000)` value) and `BOOTMEM_ALLOC_ACCESSIBLE` (equal to the current value of the `memblock.current_limit`) as minimum address of the memory region and maximum address of the memory region.

Let's look on the implementation of the `setup_command_line`:

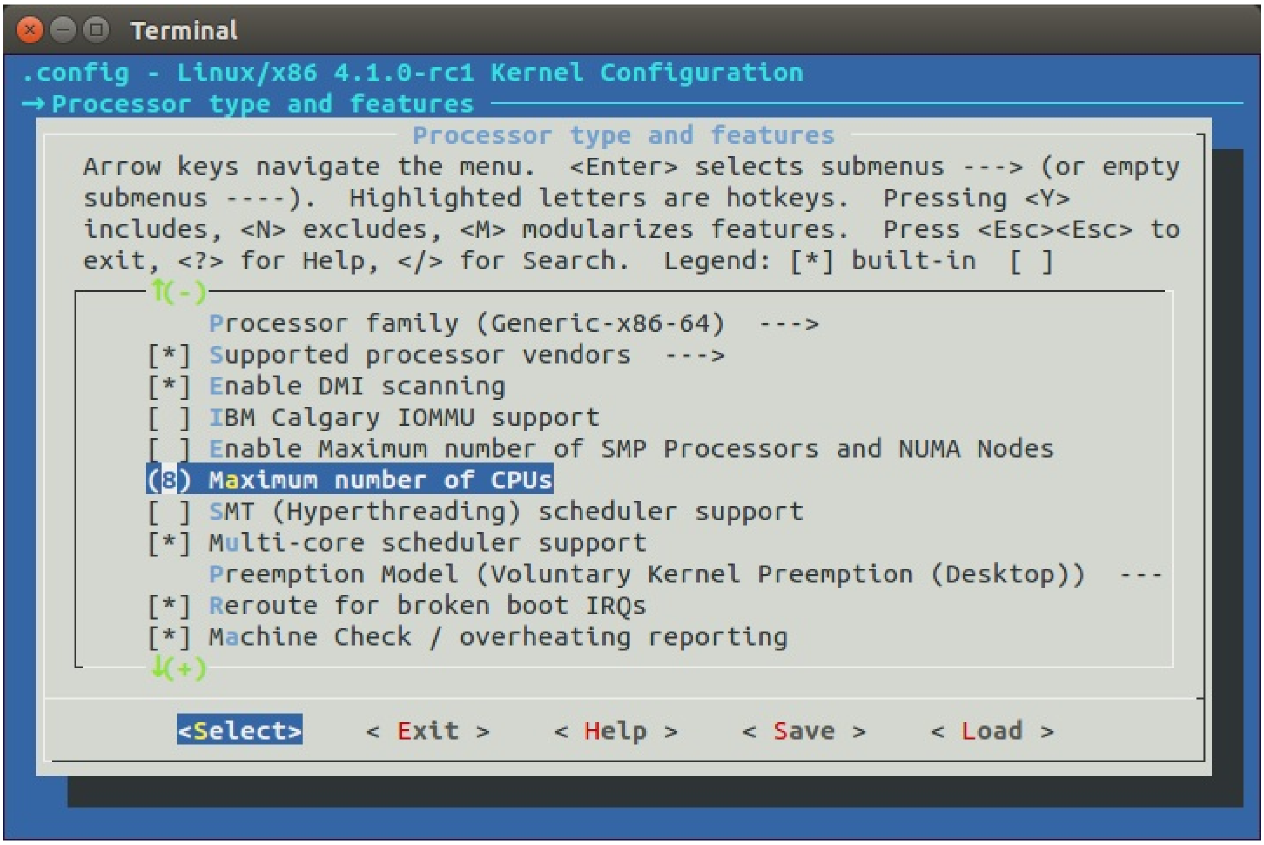
```
static void __init setup_command_line(char *command_line)
{
        saved_command_line =
                memblock_virt_alloc(strlen(boot_command_line) + 1, 0);
        initcall_command_line =
                memblock_virt_alloc(strlen(boot_command_line) + 1, 0);
        static_command_line = memblock_virt_alloc(strlen(command_line) + 1, 0);
        strcpy(saved_command_line, boot_command_line);
        strcpy(static_command_line, command_line);
 }
```

Here we can see that we allocate space for the three buffers which will contain kernel command line for the different purposes (read above). And as we allocated space, we store `boot_command_line` in the `saved_command_line` and `command_line` (kernel command line from the `setup_arch`) to the `static_command_line`.

The next function after the `setup_command_line` is the `setup_nr_cpu_ids`. This function setting `nr_cpu_ids` (number of CPUs) according to the last bit in the `cpu_possible_mask` (more about it you can read in the chapter describes cpumasks concept). Let's look on its implementation:

```
void __init setup_nr_cpu_ids(void)
{
        nr_cpu_ids = find_last_bit(cpumask_bits(cpu_possible_mask),NR_CPUS) + 1;
}
```

Here `nr_cpu_ids` represents number of CPUs, `NR_CPUS` represents the maximum number of CPUs which we can set in configuration time:



Actually we need to call this function, because `NR_CPUS` can be greater than actual amount of the CPUs in the your computer. Here we can see that we call `find_last_bit` function and pass two parameters to it:

- `cpu_possible_mask` bits;
- maximum number of CPUS.

In the `setup_arch` we can find the call of the `prefill_possible_map` function which calculates and writes to the `cpu_possible_mask` actual number of the CPUs. We call the `find_last_bit` function which takes the address and maximum size to search and returns bit number of the first set bit. We passed `cpu_possible_mask` bits and maximum number of the CPUs. First of all the `find_last_bit` function splits given `unsigned long` address to the words:

```
words = size / BITS_PER_LONG;
```

where `BITS_PER_LONG` is `64` on the `x86_64`. As we got amount of words in the given size of the search data, we need to check is given size does not contain partial words with the following check:

```
if (size & (BITS_PER_LONG-1)) {
        tmp = (addr[words] & (~0UL >> (BITS_PER_LONG
                                 - (size & (BITS_PER_LONG-1)))));
        if (tmp)
                goto found;
}
```

if it contains partial word, we mask the last word and check it. If the last word is not zero, it means that current word contains at least one set bit. We go to the `found` label:

```
found:
    return words * BITS_PER_LONG + __fls(tmp);
```

Here you can see `__fls` function which returns last set bit in a given word with help of the `bsr` instruction:

```
static inline unsigned long __fls(unsigned long word)
{
        asm("bsr %1,%0"
            : "=r" (word)
            : "rm" (word));
        return word;
}
```

The `bsr` instruction which scans the given operand for first bit set. If the last word is not partial we going through the all words in the given address and trying to find first set bit:

```
while (words) {
    tmp = addr[--words];
    if (tmp) {
found:
        return words * BITS_PER_LONG + __fls(tmp);
    }
}
```

Here we put the last word to the `tmp` variable and check that `tmp` contains at least one set bit. If a set bit found, we return the number of this bit. If no one words do not contains set bit we just return given size:

```
return size;
```

After this `nr_cpu_ids` will contain the correct amount of the available CPUs.

That's all.

Conclusion

It is the end of the seventh part about the linux kernel initialization process. In this part, finally we have finished with the `setup_arch` function and returned to the `start_kernel` function. In the next part we will continue to learn generic kernel code from the `start_kernel` and will continue our way to the first `init` process.

If you have any questions or suggestions write me a comment or ping me at twitter.

Please note that English is not my first language, And I am really sorry for any inconvenience. If you find any mistakes please send me PR to linux-insides.

Links

- Desktop Management Interface
- x86_64
- initrd
- Kernel panic
- Documentation/kernel-parameters.txt
- ACPI
- Direct memory access
- NUMA
- Control register
- vsyscalls
- SMP
- jiffy
- Previous part

Kernel initialization. Part 8.

Scheduler initialization

This is the eighth part of the Linux kernel initialization process and we stopped on the `setup_nr_cpu_ids` function in the previous part. The main point of the current part is scheduler initialization. But before we will start to learn initialization process of the scheduler, we need to do some stuff. The next step in the init/main.c is the `setup_per_cpu_areas` function. This function setups areas for the `percpu` variables, more about it you can read in the special part about the Per-CPU variables. After `percpu` areas is up and running, the next step is the `smp_prepare_boot_cpu` function. This function does some preparations for the SMP:

```
static inline void smp_prepare_boot_cpu(void)
{
         smp_ops.smp_prepare_boot_cpu();
}
```

where the `smp_prepare_boot_cpu` expands to the call of the `native_smp_prepare_boot_cpu` function (more about `smp_ops` will be in the special parts about `SMP`):

```
void __init native_smp_prepare_boot_cpu(void)
{
        int me = smp_processor_id();
        switch_to_new_gdt(me);
        cpumask_set_cpu(me, cpu_callout_mask);
        per_cpu(cpu_state, me) = CPU_ONLINE;
}
```

The `native_smp_prepare_boot_cpu` function gets the id of the current CPU (which is Bootstrap processor and its `id` is zero) with the `smp_processor_id` function. I will not explain how the `smp_processor_id` works, because we already saw it in the Kernel entry point part. As we got processor `id` number we reload Global Descriptor Table for the given CPU with the `switch_to_new_gdt` function:

```
void switch_to_new_gdt(int cpu)
{
        struct desc_ptr gdt_descr;

        gdt_descr.address = (long)get_cpu_gdt_table(cpu);
        gdt_descr.size = GDT_SIZE - 1;
        load_gdt(&gdt_descr);
        load_percpu_segment(cpu);
}
```

The `gdt_descr` variable represents pointer to the `GDT` descriptor here (we already saw `desc_ptr` in the Early interrupt and exception handling). We get the address and the size of the `GDT` descriptor where `GDT_SIZE` is `256` or:

```
#define GDT_SIZE (GDT_ENTRIES * 8)
```

and the address of the descriptor we will get with the `get_cpu_gdt_table` :

```
static inline struct desc_struct *get_cpu_gdt_table(unsigned int cpu)
{
        return per_cpu(gdt_page, cpu).gdt;
}
```

The `get_cpu_gdt_table` uses `per_cpu` macro for getting `gdt_page` percpu variable for the given CPU number (bootstrap processor with `id` - 0 in our case). You may ask the following question: so, if we can access `gdt_page` percpu variable, where it was defined? Actually we already saw it in this book. If you have read the first part of this chapter, you can remember that we saw definition of the `gdt_page` in the arch/x86/kernel/head_64.S:

```
early_gdt_descr:
    .word    GDT_ENTRIES*8-1
early_gdt_descr_base:
    .quad    INIT_PER_CPU_VAR(gdt_page)
```

and if we will look on the linker file we can see that it locates after the `__per_cpu_load` symbol:

```
#define INIT_PER_CPU(x) init_per_cpu__##x = x + __per_cpu_load
INIT_PER_CPU(gdt_page);
```

and filled `gdt_page` in the arch/x86/kernel/cpu/common.c:

```
DEFINE_PER_CPU_PAGE_ALIGNED(struct gdt_page, gdt_page) = { .gdt = {
#ifdef CONFIG_X86_64
    [GDT_ENTRY_KERNEL32_CS]        = GDT_ENTRY_INIT(0xc09b, 0, 0xfffff),
    [GDT_ENTRY_KERNEL_CS]        = GDT_ENTRY_INIT(0xa09b, 0, 0xfffff),
    [GDT_ENTRY_KERNEL_DS]        = GDT_ENTRY_INIT(0xc093, 0, 0xfffff),
    [GDT_ENTRY_DEFAULT_USER32_CS]    = GDT_ENTRY_INIT(0xc0fb, 0, 0xfffff),
    [GDT_ENTRY_DEFAULT_USER_DS]    = GDT_ENTRY_INIT(0xc0f3, 0, 0xfffff),
    [GDT_ENTRY_DEFAULT_USER_CS]    = GDT_ENTRY_INIT(0xa0fb, 0, 0xfffff),
    ...
    ...
    ...
```

more about `percpu` variables you can read in the Per-CPU variables part. As we got address and size of the `GDT` descriptor we reload `GDT` with the `load_gdt` which just execute `lgdt` instruct and load `percpu_segment` with the following function:

```
void load_percpu_segment(int cpu) {
    loadsegment(gs, 0);
    wrmsrl(MSR_GS_BASE, (unsigned long)per_cpu(irq_stack_union.gs_base, cpu));
    load_stack_canary_segment();
}
```

The base address of the `percpu` area must contain `gs` register (or `fs` register for `x86`), so we are using `loadsegment` macro and pass `gs` . In the next step we writes the base address if the IRQ stack and setup stack canary (this is only for `x86_32`). After we load new `GDT` , we fill `cpu_callout_mask` bitmap with the current cpu and set cpu state as online with the setting `cpu_state` percpu variable for the current processor - `CPU_ONLINE` :

```
cpumask_set_cpu(me, cpu_callout_mask);
per_cpu(cpu_state, me) = CPU_ONLINE;
```

So, what is `cpu_callout_mask` bitmap... As we initialized bootstrap processor (processor which is booted the first on `x86`) the other processors in a multiprocessor system are known as `secondary processors` . Linux kernel uses following two bitmasks:

- `cpu_callout_mask`
- `cpu_callin_mask`

After bootstrap processor initialized, it updates the `cpu_callout_mask` to indicate which secondary processor can be initialized next. All other or secondary processors can do some initialization stuff before and check the `cpu_callout_mask` on the boostrap processor bit. Only after the bootstrap processor filled the `cpu_callout_mask` with this secondary processor, it will continue the rest of its initialization. After that the certain processor finish its initialization process, the processor sets bit in the `cpu_callin_mask` . Once the bootstrap

processor finds the bit in the `cpu_callin_mask` for the current secondary processor, this processor repeats the same procedure for initialization of one of the remaining secondary processors. In a short words it works as i described, but we will see more details in the chapter about `SMP`.

That's all. We did all `SMP` boot preparation.

Build zonelists

In the next step we can see the call of the `build_all_zonelists` function. This function sets up the order of zones that allocations are preferred from. What are zones and what's order we will understand soon. For the start let's see how linux kernel considers physical memory. Physical memory is split into banks which are called - `nodes`. If you has no hardware support for `NUMA`, you will see only one node:

```
$ cat /sys/devices/system/node/node0/numastat
numa_hit 72452442
numa_miss 0
numa_foreign 0
interleave_hit 12925
local_node 72452442
other_node 0
```

Every `node` is presented by the `struct pglist_data` in the linux kernel. Each node is divided into a number of special blocks which are called - `zones`. Every zone is presented by the `zone struct` in the linux kernel and has one of the type:

- `ZONE_DMA` - 0-16M;
- `ZONE_DMA32` - used for 32 bit devices that can only do DMA areas below 4G;
- `ZONE_NORMAL` - all RAM from the 4GB on the `x86_64`;
- `ZONE_HIGHMEM` - absent on the `x86_64`;
- `ZONE_MOVABLE` - zone which contains movable pages.

which are presented by the `zone_type` enum. We can get information about zones with the:

```
$ cat /proc/zoneinfo
Node 0, zone      DMA
  pages free     3975
        min      3
        low      3
        ...
        ...
Node 0, zone    DMA32
  pages free     694163
        min      875
        low      1093
        ...
        ...
Node 0, zone   Normal
  pages free     2529995
        min      3146
        low      3932
        ...
        ...
```

As I wrote above all nodes are described with the `pglist_data` or `pg_data_t` structure in memory. This structure is defined in the include/linux/mmzone.h. The `build_all_zonelists` function from the mm/page_alloc.c constructs an ordered `zonelist` (of different zones `DMA`, `DMA32`, `NORMAL`, `HIGH_MEMORY`, `MOVABLE`) which specifies the zones/nodes to visit when a selected `zone` or `node` cannot satisfy the allocation request. That's all. More about `NUMA` and multiprocessor systems will be in the special part.

The rest of the stuff before scheduler initialization

Before we will start to dive into linux kernel scheduler initialization process we must do a couple of things. The first thing is the `page_alloc_init` function from the mm/page_alloc.c. This function looks pretty easy:

```
void __init page_alloc_init(void)
{
        hotcpu_notifier(page_alloc_cpu_notify, 0);
}
```

and initializes handler for the `CPU` hotplug. Of course the `hotcpu_notifier` depends on the `CONFIG_HOTPLUG_CPU` configuration option and if this option is set, it just calls `cpu_notifier` macro which expands to the call of the `register_cpu_notifier` which adds hotplug cpu handler (`page_alloc_cpu_notify` in our case).

After this we can see the kernel command line in the initialization output:

```
Linux version 4.1.0-rc2+ (alex@localhost) (gcc version 4.9.2 (Ubuntu 4.9.2-10ubuntu13) ) #493 SMP Thu
Command line: root=/dev/sdb earlyprintk=ttyS0,115200 loglevel=7 debug rdinit=/sbin/init root=/dev/ram
```

And a couple of functions such as `parse_early_param` and `parse_args` which handles linux kernel command line. You may remember that we already saw the call of the `parse_early_param` function in the sixth part of the kernel initialization chapter, so why we call it again? Answer is simple: we call this function in the architecture-specific code (`x86_64` in our case), but not all architecture calls this function. And we need to call the second function `parse_args` to parse and handle non-early command line arguments.

In the next step we can see the call of the `jump_label_init` from the kernel/jump_label.c. and initializes jump label.

After this we can see the call of the `setup_log_buf` function which setups the printk log buffer. We already saw this function in the seventh part of the linux kernel initialization process chapter.

PID hash initialization

The next is `pidhash_init` function. As you know each process has assigned a unique number which called - `process identification number` or `PID`. Each process generated with fork or clone is automatically assigned a new unique `PID` value by the kernel. The management of `PIDs` centered around the two special data structures: `struct pid` and `struct upid`. First structure represents information about a `PID` in the kernel. The second structure represents the information that is visible in a specific namespace. All `PID` instances stored in the special hash table:

```
static struct hlist_head *pid_hash;
```

This hash table is used to find the pid instance that belongs to a numeric `PID` value. So, `pidhash_init` initializes this hash table. In the start of the `pidhash_init` function we can see the call of the `alloc_large_system_hash`:

```
pid_hash = alloc_large_system_hash("PID", sizeof(*pid_hash), 0, 18,
                                   HASH_EARLY | HASH_SMALL,
                                   &pidhash_shift, NULL,
                                   0, 4096);
```

The number of elements of the `pid_hash` depends on the `RAM` configuration, but it can be between `2^4` and `2^12`. The `pidhash_init` computes the size and allocates the required storage (which is `hlist` in our case - the same as doubly linked list, but contains one

pointer instead on the struct hlist_head]. The `alloc_large_system_hash` function allocates a large system hash table with `memblock_virt_alloc_nopanic` if we pass `HASH_EARLY` flag (as it in our case) or with `__vmalloc` if we did no pass this flag.

The result we can see in the `dmesg` output:

```
$ dmesg | grep hash
[    0.000000] PID hash table entries: 4096 (order: 3, 32768 bytes)
...
...
...
```

That's all. The rest of the stuff before scheduler initialization is the following functions: `vfs_caches_init_early` does early initialization of the virtual file system (more about it will be in the chapter which will describe virtual file system), `sort_main_extable` sorts the kernel's built-in exception table entries which are between `__start___ex_table` and `__stop___ex_table`, and `trap_init` initializes trap handlers (more about last two function we will know in the separate chapter about interrupts).

The last step before the scheduler initialization is initialization of the memory manager with the `mm_init` function from the init/main.c. As we can see, the `mm_init` function initializes different parts of the linux kernel memory manager:

```
page_ext_init_flatmem();
mem_init();
kmem_cache_init();
percpu_init_late();
pgtable_init();
vmalloc_init();
```

The first is `page_ext_init_flatmem` which depends on the `CONFIG_SPARSEMEM` kernel configuration option and initializes extended data per page handling. The `mem_init` releases all `bootmem`, the `kmem_cache_init` initializes kernel cache, the `percpu_init_late` - replaces `percpu` chunks with those allocated by slub, the `pgtable_init` - initializes the `page->ptl` kernel cache, the `vmalloc_init` - initializes `vmalloc`. Please, **NOTE** that we will not dive into details about all of these functions and concepts, but we will see all of they it in the Linux kernel memory manager chapter.

That's all. Now we can look on the `scheduler`.

Scheduler initialization

And now we come to the main purpose of this part - initialization of the task scheduler. I want to say again as I already did it many times, you will not see the full explanation of the scheduler here, there will be special chapter about this. Ok, next point is the `sched_init` function from the kernel/sched/core.c and as we can understand from the function's name, it initializes scheduler. Let's start to dive into this function and try to understand how the scheduler is initialized. At the start of the `sched_init` function we can see the following code:

```
#ifdef CONFIG_FAIR_GROUP_SCHED
         alloc_size += 2 * nr_cpu_ids * sizeof(void **);
#endif
#ifdef CONFIG_RT_GROUP_SCHED
         alloc_size += 2 * nr_cpu_ids * sizeof(void **);
#endif
```

First of all we can see two configuration options here:

- `CONFIG_FAIR_GROUP_SCHED`
- `CONFIG_RT_GROUP_SCHED`

Both of this options provide two different planning models. As we can read from the documentation, the current scheduler - `CFS` or `Completely Fair Scheduler` use a simple concept. It models process scheduling as if the system has an ideal multitasking processor where each process would receive `1/n` processor time, where `n` is the number of the runnable processes. The scheduler uses the special set of rules. These rules determine when and how to select a new process to run and they are called `scheduling policy`. The Completely Fair Scheduler supports following `normal` or `non-real-time` scheduling policies: `SCHED_NORMAL`, `SCHED_BATCH` and `SCHED_IDLE`. The `SCHED_NORMAL` is used for the most normal applications, the amount of cpu each process consumes is mostly determined by the nice value, the `SCHED_BATCH` used for the 100% non-interactive tasks and the `SCHED_IDLE` runs tasks only when the processor has no task to run besides this task. The `real-time` policies are also supported for the time-critical applications: `SCHED_FIFO` and `SCHED_RR`. If you've read something about the Linux kernel scheduler, you can know that it is modular. It means that it supports different algorithms to schedule different types of processes. Usually this modularity is called `scheduler classes`. These modules encapsulate scheduling policy details and are handled by the scheduler core without knowing too much about them.

Now let's back to the our code and look on the two configuration options `CONFIG_FAIR_GROUP_SCHED` and `CONFIG_RT_GROUP_SCHED`. The scheduler operates on an individual task. These options allows to schedule group tasks (more about it you can read in

the CFS group scheduling). We can see that we assign the `alloc_size` variables which represent size based on amount of the processors to allocate for the `sched_entity` and `cfs_rq` to the `2 * nr_cpu_ids * sizeof(void **)` expression with `kzalloc`:

```
ptr = (unsigned long)kzalloc(alloc_size, GFP_NOWAIT);

#ifdef CONFIG_FAIR_GROUP_SCHED
        root_task_group.se = (struct sched_entity **)ptr;
        ptr += nr_cpu_ids * sizeof(void **);

        root_task_group.cfs_rq = (struct cfs_rq **)ptr;
        ptr += nr_cpu_ids * sizeof(void **);
#endif
```

The `sched_entity` is a structure which is defined in the include/linux/sched.h and used by the scheduler to keep track of process accounting. The `cfs_rq` presents run queue. So, you can see that we allocated space with size `alloc_size` for the run queue and scheduler entity of the `root_task_group`. The `root_task_group` is an instance of the `task_group` structure from the kernel/sched/sched.h which contains task group related information:

```
struct task_group {
    ...
    ...
    struct sched_entity **se;
    struct cfs_rq **cfs_rq;
    ...
    ...
}
```

The root task group is the task group which belongs to every task in system. As we allocated space for the root task group scheduler entity and runqueue, we go over all possible CPUs (`cpu_possible_mask` bitmap) and allocate zeroed memory from a particular memory node with the `kzalloc_node` function for the `load_balance_mask` `percpu` variable:

```
DECLARE_PER_CPU(cpumask_var_t, load_balance_mask);
```

Here `cpumask_var_t` is the `cpumask_t` with one difference: `cpumask_var_t` is allocated only `nr_cpu_ids` bits when the `cpumask_t` always has `NR_CPUS` bits (more about `cpumask` you can read in the CPU masks part). As you can see:

```
#ifdef CONFIG_CPUMASK_OFFSTACK
    for_each_possible_cpu(i) {
        per_cpu(load_balance_mask, i) = (cpumask_var_t)kzalloc_node(
                cpumask_size(), GFP_KERNEL, cpu_to_node(i));
    }
#endif
```

this code depends on the `CONFIG_CPUMASK_OFFSTACK` configuration option. This configuration options says to use dynamic allocation for `cpumask`, instead of putting it on the stack. All groups have to be able to rely on the amount of CPU time. With the call of the two following functions:

```
init_rt_bandwidth(&def_rt_bandwidth,
                  global_rt_period(), global_rt_runtime());
init_dl_bandwidth(&def_dl_bandwidth,
                  global_rt_period(), global_rt_runtime());
```

we initialize bandwidth management for the `SCHED_DEADLINE` real-time tasks. These functions initializes `rt_bandwidth` and `dl_bandwidth` structures which store information about maximum `deadline` bandwidth of the system. For example, let's look on the implementation of the `init_rt_bandwidth` function:

```
void init_rt_bandwidth(struct rt_bandwidth *rt_b, u64 period, u64 runtime)
{
        rt_b->rt_period = ns_to_ktime(period);
        rt_b->rt_runtime = runtime;

        raw_spin_lock_init(&rt_b->rt_runtime_lock);

        hrtimer_init(&rt_b->rt_period_timer,
                     CLOCK_MONOTONIC, HRTIMER_MODE_REL);
        rt_b->rt_period_timer.function = sched_rt_period_timer;
}
```

It takes three parameters:

- address of the `rt_bandwidth` structure which contains information about the allocated and consumed quota within a period;
- `period` - period over which real-time task bandwidth enforcement is measured in `us`;
- `runtime` - part of the period that we allow tasks to run in `us`.

As `period` and `runtime` we pass result of the `global_rt_period` and `global_rt_runtime` functions. Which are `1s` second and and `0.95s` by default. The `rt_bandwidth` structure is defined in the kernel/sched/sched.h and looks:

```
struct rt_bandwidth {
        raw_spinlock_t          rt_runtime_lock;
        ktime_t                 rt_period;
        u64                     rt_runtime;
        struct hrtimer          rt_period_timer;
};
```

As you can see, it contains `runtime` and `period` and also two following fields:

- `rt_runtime_lock` - spinlock for the `rt_time` protection;
- `rt_period_timer` - high-resolution kernel timer for unthrottled of real-time tasks.

So, in the `init_rt_bandwidth` we initialize `rt_bandwidth` period and runtime with the given parameters, initialize the spinlock and high-resolution time. In the next step, depends on enable of SMP, we make initialization of the root domain:

```
#ifdef CONFIG_SMP
    init_defrootdomain();
#endif
```

The real-time scheduler requires global resources to make scheduling decision. But unfortunately scalability bottlenecks appear as the number of CPUs increase. The concept of root domains was introduced for improving scalability. The linux kernel provides a special mechanism for assigning a set of CPUs and memory nodes to a set of tasks and it is called - `cpuset`. If a `cpuset` contains non-overlapping with other `cpuset` CPUs, it is `exclusive cpuset`. Each exclusive cpuset defines an isolated domain or `root domain` of CPUs partitioned from other cpusets or CPUs. A `root domain` is presented by the `struct root_domain` from the kernel/sched/sched.h in the linux kernel and its main purpose is to narrow the scope of the global variables to per-domain variables and all real-time scheduling decisions are made only within the scope of a root domain. That's all about it, but we will see more details about it in the chapter about real-time scheduler.

After `root domain` initialization, we make initialization of the bandwidth for the real-time tasks of the root task group as we did it above:

```
#ifdef CONFIG_RT_GROUP_SCHED
    init_rt_bandwidth(&root_task_group.rt_bandwidth,
            global_rt_period(), global_rt_runtime());
#endif
```

In the next step, depends on the `CONFIG_CGROUP_SCHED` kernel configuration option we initialize the `siblings` and `children` lists of the root task group. As we can read from the documentation, the `CONFIG_CGROUP_SCHED` is:

```
This option allows you to create arbitrary task groups using the "cgroup" pseudo
filesystem and control the cpu bandwidth allocated to each such task group.
```

As we finished with the lists initialization, we can see the call of the `autogroup_init` function:

```
#ifdef CONFIG_CGROUP_SCHED
         list_add(&root_task_group.list, &task_groups);
         INIT_LIST_HEAD(&root_task_group.children);
         INIT_LIST_HEAD(&root_task_group.siblings);
         autogroup_init(&init_task);
#endif
```

which initializes automatic process group scheduling.

After this we are going through the all `possible` cpu (you can remember that `possible` CPUs store in the `cpu_possible_mask` bitmap that can ever be available in the system) and initialize a `runqueue` for each possible cpu:

```
for_each_possible_cpu(i) {
    struct rq *rq;
    ...
    ...
    ...
```

Each processor has its own locking and individual runqueue. All runnable tasks are stored in an active array and indexed according to its priority. When a process consumes its time slice, it is moved to an expired array. All of these arras are stored in the special structure which names is `runqueue`. As there are no global lock and runqueue, we are going through the all possible CPUs and initialize runqueue for the every cpu. The `runqueue` is presented by the `rq` structure in the linux kernel which is defined in the kernel/sched/sched.h.

```
rq = cpu_rq(i);
raw_spin_lock_init(&rq->lock);
rq->nr_running = 0;
rq->calc_load_active = 0;
rq->calc_load_update = jiffies + LOAD_FREQ;
init_cfs_rq(&rq->cfs);
init_rt_rq(&rq->rt);
init_dl_rq(&rq->dl);
rq->rt.rt_runtime = def_rt_bandwidth.rt_runtime;
```

Here we get the runqueue for the every CPU with the `cpu_rq` macro which returns `runqueues` percpu variable and start to initialize it with runqueue lock, number of running tasks, `calc_load` relative fields (`calc_load_active` and `calc_load_update`) which are used in the reckoning of a CPU load and initialization of the completely fair, real-time and deadline related fields in a runqueue. After this we initialize `cpu_load` array with zeros and set the last load update tick to the `jiffies` variable which determines the number of time ticks (cycles), since the system boot:

```
for (j = 0; j < CPU_LOAD_IDX_MAX; j++)
    rq->cpu_load[j] = 0;

rq->last_load_update_tick = jiffies;
```

where `cpu_load` keeps history of runqueue loads in the past, for now `CPU_LOAD_IDX_MAX` is 5. In the next step we fill `runqueue` fields which are related to the SMP, but we will not cover them in this part. And in the end of the loop we initialize high-resolution timer for the give `runqueue` and set the `iowait` (more about it in the separate part about scheduler) number:

```
init_rq_hrtick(rq);
atomic_set(&rq->nr_iowait, 0);
```

Now we come out from the `for_each_possible_cpu` loop and the next we need to set load weight for the `init` task with the `set_load_weight` function. Weight of process is calculated through its dynamic priority which is static priority + scheduling class of the process. After this we increase memory usage counter of the memory descriptor of the `init` process and set scheduler class for the current process:

```
atomic_inc(&init_mm.mm_count);
current->sched_class = &fair_sched_class;
```

And make current process (it will be the first `init` process) `idle` and update the value of the `calc_load_update` with the 5 seconds interval:

```
init_idle(current, smp_processor_id());
calc_load_update = jiffies + LOAD_FREQ;
```

So, the `init` process will be run, when there will be no other candidates (as it is the first process in the system). In the end we just set `scheduler_running` variable:

```
scheduler_running = 1;
```

That's all. Linux kernel scheduler is initialized. Of course, we have skipped many different details and explanations here, because we need to know and understand how different concepts (like process and process groups, runqueue, rcu, etc.) works in the linux kernel , but we took a short look on the scheduler initialization process. We will look all other details in the separate part which will be fully dedicated to the scheduler.

Conclusion

It is the end of the eighth part about the linux kernel initialization process. In this part, we looked on the initialization process of the scheduler and we will continue in the next part to dive in the linux kernel initialization process and will see initialization of the RCU and many other initialization stuff in the next part.

If you have any questions or suggestions write me a comment or ping me at twitter.

Please note that English is not my first language, And I am really sorry for any inconvenience. If you find any mistakes please send me PR to linux-insides.

Links

- CPU masks
- high-resolution kernel timer
- spinlock
- Run queue
- Linux kernem memory manager
- slub
- virtual file system
- Linux kernel hotplug documentation
- IRQ
- Global Descriptor Table
- Per-CPU variables
- SMP
- RCU
- CFS Scheduler documentation
- Real-Time group scheduling
- Previous part

Kernel initialization. Part 9.

RCU initialization

This is ninth part of the Linux Kernel initialization process and in the previous part we stopped at the scheduler initialization. In this part we will continue to dive to the linux kernel initialization process and the main purpose of this part will be to learn about initialization of the RCU. We can see that the next step in the init/main.c after the `sched_init` is the call of the `preempt_disable`. There are two macros:

- `preempt_disable`
- `preempt_enable`

for preemption disabling and enabling. First of all let's try to understand what is `preempt` in the context of an operating system kernel. In simple words, preemption is ability of the operating system kernel to preempt current task to run task with higher priority. Here we need to disable preemption because we will have only one `init` process for the early boot time and we don't need to stop it before we call `cpu_idle` function. The `preempt_disable` macro is defined in the include/linux/preempt.h and depends on the `CONFIG_PREEMPT_COUNT` kernel configuration option. This macro is implemented as:

```
#define preempt_disable() \
do { \
        preempt_count_inc(); \
        barrier(); \
} while (0)
```

and if `CONFIG_PREEMPT_COUNT` is not set just:

```
#define preempt_disable()                       barrier()
```

Let's look on it. First of all we can see one difference between these macro implementations. The `preempt_disable` with `CONFIG_PREEMPT_COUNT` set contains the call of the `preempt_count_inc`. There is special `percpu` variable which stores the number of held locks and `preempt_disable` calls:

```
DECLARE_PER_CPU(int, __preempt_count);
```

In the first implementation of the `preempt_disable` we increment this `__preempt_count`. There is API for returning value of the `__preempt_count`, it is the `preempt_count` function. As we called `preempt_disable`, first of all we increment preemption counter with the `preempt_count_inc` macro which expands to the:

```
#define preempt_count_inc() preempt_count_add(1)
#define preempt_count_add(val)  __preempt_count_add(val)
```

where `preempt_count_add` calls the `raw_cpu_add_4` macro which adds `1` to the given `percpu` variable (`__preempt_count`) in our case (more about `precpu` variables you can read in the part about Per-CPU variables). Ok, we increased `__preempt_count` and the next step we can see the call of the `barrier` macro in the both macros. The `barrier` macro inserts an optimization barrier. In the processors with `x86_64` architecture independent memory access operations can be performed in any order. That's why we need the opportunity to point compiler and processor on compliance of order. This mechanism is memory barrier. Let's consider a simple example:

```
preempt_disable();
foo();
preempt_enable();
```

Compiler can rearrange it as:

```
preempt_disable();
preempt_enable();
foo();
```

In this case non-preemptible function `foo` can be preempted. As we put `barrier` macro in the `preempt_disable` and `preempt_enable` macros, it prevents the compiler from swapping `preempt_count_inc` with other statements. More about barriers you can read here and here.

In the next step we can see following statement:

```
if (WARN(!irqs_disabled(),
     "Interrupts were enabled *very* early, fixing it\n"))
    local_irq_disable();
```

which check IRQs state, and disabling (with `cli` instruction for `x86_64`) if they are enabled.

That's all. Preemption is disabled and we can go ahead.

Initialization of the integer ID management

In the next step we can see the call of the `idr_init_cache` function which defined in the lib/idr.c. The `idr` library is used in a various places in the linux kernel to manage assigning integer `IDs` to objects and looking up objects by id.

Let's look on the implementation of the `idr_init_cache` function:

```
void __init idr_init_cache(void)
{
        idr_layer_cache = kmem_cache_create("idr_layer_cache",
                                sizeof(struct idr_layer), 0, SLAB_PANIC, NULL);
}
```

Here we can see the call of the `kmem_cache_create`. We already called the `kmem_cache_init` in the init/main.c. This function create generalized caches again using the `kmem_cache_alloc` (more about caches we will see in the Linux kernel memory management chapter). In our case, as we are using `kmem_cache_t` which will be used by the slab allocator and `kmem_cache_create` creates it. As you can see we pass five parameters to the `kmem_cache_create`:

- name of the cache;
- size of the object to store in cache;
- offset of the first object in the page;
- flags;
- constructor for the objects.

and it will create `kmem_cache` for the integer IDs. Integer `IDs` is commonly used pattern to map set of integer IDs to the set of pointers. We can see usage of the integer IDs in the i2c drivers subsystem. For example drivers/i2c/i2c-core.c which represents the core of the `i2c` subsystem defines `ID` for the `i2c` adapter with the `DEFINE_IDR` macro:

```
static DEFINE_IDR(i2c_adapter_idr);
```

and then uses it for the declaration of the `i2c` adapter:

```
static int __i2c_add_numbered_adapter(struct i2c_adapter *adap)
{
  int     id;
  ...
  ...
  ...
  id = idr_alloc(&i2c_adapter_idr, adap, adap->nr, adap->nr + 1, GFP_KERNEL);
  ...
  ...
  ...
}
```

and `id2_adapter_idr` presents dynamically calculated bus number.

More about integer ID management you can read here.

RCU initialization

The next step is RCU initialization with the `rcu_init` function and it's implementation depends on two kernel configuration options:

- `CONFIG_TINY_RCU`
- `CONFIG_TREE_RCU`

In the first case `rcu_init` will be in the kernel/rcu/tiny.c and in the second case it will be defined in the kernel/rcu/tree.c. We will see the implementation of the `tree rcu`, but first of all about the `RCU` in general.

`RCU` or read-copy update is a scalable high-performance synchronization mechanism implemented in the Linux kernel. On the early stage the linux kernel provided support and environment for the concurrently running applications, but all execution was serialized in the kernel using a single global lock. In our days linux kernel has no single global lock, but provides different mechanisms including lock-free data structures, percpu data structures and other. One of these mechanisms is - the `read-copy update`. The `RCU` technique is designed for rarely-modified data structures. The idea of the `RCU` is simple. For example we have a rarely-modified data structure. If somebody wants to change this data structure, we make a copy of this data structure and make all changes in the copy. In the same time all other users of the data structure use old version of it. Next, we need to choose safe moment when original version of the data structure will have no users and update it with the modified copy.

Of course this description of the `RCU` is very simplified. To understand some details about `RCU`, first of all we need to learn some terminology. Data readers in the `RCU` executed in the critical section. Every time when data reader get to the critical section, it calls the

`rcu_read_lock` , and `rcu_read_unlock` on exit from the critical section. If the thread is not in the critical section, it will be in state which called - `quiescent state` . The moment when every thread is in the `quiescent state` called - `grace period` . If a thread wants to remove an element from the data structure, this occurs in two steps. First step is `removal` - atomically removes element from the data structure, but does not release the physical memory. After this thread-writer announces and waits until it is finished. From this moment, the removed element is available to the thread-readers. After the `grace period` finished, the second step of the element removal will be started, it just removes the element from the physical memory.

There a couple of implementations of the `RCU` . Old `RCU` called classic, the new implementation called `tree` RCU. As you may already understand, the `CONFIG_TREE_RCU` kernel configuration option enables tree `RCU` . Another is the `tiny` RCU which depends on `CONFIG_TINY_RCU` and `CONFIG_SMP=n` . We will see more details about the `RCU` in general in the separate chapter about synchronization primitives, but now let's look on the `rcu_init` implementation from the kernel/rcu/tree.c:

```
void __init rcu_init(void)
{
        int cpu;

        rcu_bootup_announce();
        rcu_init_geometry();
        rcu_init_one(&rcu_bh_state, &rcu_bh_data);
        rcu_init_one(&rcu_sched_state, &rcu_sched_data);
        __rcu_init_preempt();
        open_softirq(RCU_SOFTIRQ, rcu_process_callbacks);

        /*
         * We don't need protection against CPU-hotplug here because
         * this is called early in boot, before either interrupts
         * or the scheduler are operational.
         */
        cpu_notifier(rcu_cpu_notify, 0);
        pm_notifier(rcu_pm_notify, 0);
        for_each_online_cpu(cpu)
                rcu_cpu_notify(NULL, CPU_UP_PREPARE, (void *)(long)cpu);

        rcu_early_boot_tests();
}
```

In the beginning of the `rcu_init` function we define `cpu` variable and call `rcu_bootup_announce` . The `rcu_bootup_announce` function is pretty simple:

```
static void __init rcu_bootup_announce(void)
{
        pr_info("Hierarchical RCU implementation.\n");
        rcu_bootup_announce_oddness();
}
```

It just prints information about the `RCU` with the `pr_info` function and `rcu_bootup_announce_oddness` which uses `pr_info` too, for printing different information about the current `RCU` configuration which depends on different kernel configuration options like `CONFIG_RCU_TRACE` , `CONFIG_PROVE_RCU` , `CONFIG_RCU_FANOUT_EXACT` , etc. In the next step, we can see the call of the `rcu_init_geometry` function. This function is defined in the same source code file and computes the node tree geometry depends on the amount of CPUs. Actually `RCU` provides scalability with extremely low internal RCU lock contention. What if a data structure will be read from the different CPUs? `RCU` API provides the `rcu_state` structure which presents RCU global state including node hierarchy. Hierarchy is presented by the:

```
struct rcu_node node[NUM_RCU_NODES];
```

array of structures. As we can read in the comment of above definition:

```
The root (first level) of the hierarchy is in ->node[0] (referenced by ->level[0]), th
e second
level in ->node[1] through ->node[m] (->node[1] referenced by ->level[1]), and the thi
rd level
in ->node[m+1] and following (->node[m+1] referenced by ->level[2]).  The number of le
vels is
determined by the number of CPUs and by CONFIG_RCU_FANOUT.

Small systems will have a "hierarchy" consisting of a single rcu_node.
```

The `rcu_node` structure is defined in the kernel/rcu/tree.h and contains information about current grace period, is grace period completed or not, CPUs or groups that need to switch in order for current grace period to proceed, etc. Every `rcu_node` contains a lock for a couple of CPUs. These `rcu_node` structures are embedded into a linear array in the `rcu_state` structure and represented as a tree with the root as the first element and covers all CPUs. As you can see the number of the rcu nodes determined by the `NUM_RCU_NODES` which depends on number of available CPUs:

```
#define NUM_RCU_NODES (RCU_SUM - NR_CPUS)
#define RCU_SUM (NUM_RCU_LVL_0 + NUM_RCU_LVL_1 + NUM_RCU_LVL_2 + NUM_RCU_LVL_3 + NUM_R
CU_LVL_4)
```

where levels values depend on the `CONFIG_RCU_FANOUT_LEAF` configuration option. For example for the simplest case, one `rcu_node` will cover two CPU on machine with the eight CPUs:

```
+-----------------------------------------------------------------+
|  rcu_state                                                      |
|                 +----------------------+                        |
|                 |         root         |                        |
|                 |       rcu_node       |                        |
|                 +----------------------+                        |
|                    |                |                           |
|               +----v-----+       +--v-------+                   |
|               |          |       |          |                   |
|               | rcu_node |       | rcu_node |                   |
|               |          |       |          |                   |
|         +------------------+     +----------------+             |
|         |                  |        |             |             |
|         |                  |        |             |             |
|    +----v-----+    +-------v--+   +-v--------+  +-v--------+    |
|    |          |    |          |   |          |  |          |    |
|    | rcu_node |    | rcu_node |   | rcu_node |  | rcu_node |    |
|    |          |    |          |   |          |  |          |    |
|    +----------+    +----------+   +----------+  +----------+    |
|         |                 |             |               |       |
|         |                 |             |               |       |
|         |                 |             |               |       |
|         |                 |             |               |       |
+---------|-----------------|-------------|---------------|-------+
          |                 |             |               |
+---------v-----------------v-------------v---------------v--------+
|                 |                |              |                |
|     CPU1        |      CPU3      |     CPU5     |     CPU7       |
|                 |                |              |                |
|     CPU2        |      CPU4      |     CPU6     |     CPU8       |
|                 |                |              |                |
+------------------------------------------------------------------+
```

So, in the `rcu_init_geometry` function we just need to calculate the total number of `rcu_node` structures. We start to do it with the calculation of the `jiffies` till to the first and next `fqs` which is `force-quiescent-state` (read above about it):

```
d = RCU_JIFFIES_TILL_FORCE_QS + nr_cpu_ids / RCU_JIFFIES_FQS_DIV;
if (jiffies_till_first_fqs == ULONG_MAX)
        jiffies_till_first_fqs = d;
if (jiffies_till_next_fqs == ULONG_MAX)
        jiffies_till_next_fqs = d;
```

where:

```
#define RCU_JIFFIES_TILL_FORCE_QS (1 + (HZ > 250) + (HZ > 500))
#define RCU_JIFFIES_FQS_DIV     256
```

As we calculated these jiffies, we check that previous defined `jiffies_till_first_fqs` and `jiffies_till_next_fqs` variables are equal to the ULONG_MAX (their default values) and set they equal to the calculated value. As we did not touch these variables before, they are equal to the `ULONG_MAX` :

```
static ulong jiffies_till_first_fqs = ULONG_MAX;
static ulong jiffies_till_next_fqs = ULONG_MAX;
```

In the next step of the `rcu_init_geometry` , we check that `rcu_fanout_leaf` didn't change (it has the same value as `CONFIG_RCU_FANOUT_LEAF` in compile-time) and equal to the value of the `CONFIG_RCU_FANOUT_LEAF` configuration option, we just return:

```
if (rcu_fanout_leaf == CONFIG_RCU_FANOUT_LEAF &&
    nr_cpu_ids == NR_CPUS)
    return;
```

After this we need to compute the number of nodes that an `rcu_node` tree can handle with the given number of levels:

```
rcu_capacity[0] = 1;
rcu_capacity[1] = rcu_fanout_leaf;
for (i = 2; i <= MAX_RCU_LVLS; i++)
    rcu_capacity[i] = rcu_capacity[i - 1] * CONFIG_RCU_FANOUT;
```

And in the last step we calculate the number of rcu_nodes at each level of the tree in the loop.

As we calculated geometry of the `rcu_node` tree, we need to go back to the `rcu_init` function and next step we need to initialize two `rcu_state` structures with the `rcu_init_one` function:

```
rcu_init_one(&rcu_bh_state, &rcu_bh_data);
rcu_init_one(&rcu_sched_state, &rcu_sched_data);
```

The `rcu_init_one` function takes two arguments:

- Global `RCU` state;
- Per-CPU data for `RCU` .

Both variables defined in the kernel/rcu/tree.h with its `percpu` data:

```
extern struct rcu_state rcu_bh_state;
DECLARE_PER_CPU(struct rcu_data, rcu_bh_data);
```

About this states you can read here. As I wrote above we need to initialize `rcu_state` structures and `rcu_init_one` function will help us with it. After the `rcu_state` initialization, we can see the call of the `__rcu_init_preempt` which depends on the `CONFIG_PREEMPT_RCU` kernel configuration option. It does the same as previous functions - initialization of the `rcu_preempt_state` structure with the `rcu_init_one` function which has `rcu_state` type. After this, in the `rcu_init` , we can see the call of the:

```
open_softirq(RCU_SOFTIRQ, rcu_process_callbacks);
```

function. This function registers a handler of the `pending interrupt` . Pending interrupt or `softirq` supposes that part of actions can be delayed for later execution when the system is less loaded. Pending interrupts is represented by the following structure:

```
struct softirq_action
{
        void    (*action)(struct softirq_action *);
};
```

which is defined in the include/linux/interrupt.h and contains only one field - handler of an interrupt. You can check about `softirqs` in the your system with the:

```
$ cat /proc/softirqs
                    CPU0       CPU1       CPU2       CPU3       CPU4       CPU5
CPU6       CPU7
          HI:          2          0          0          1          0          2
   0          0
       TIMER:     137779     108110     139573     107647     107408     114972       9
9653      98665
      NET_TX:       1127          0          4          0          1          1
   0          0
      NET_RX:        334        221     132939       3076        451        361
 292        303
       BLOCK:       5253       5596          8        779       2016      37442
  28       2855
BLOCK_IOPOLL:          0          0          0          0          0          0
   0          0
     TASKLET:         66          0       2916        113          0         24       2
6708          0
       SCHED:     102350      75950      91705      75356      75323      82627       6
9279      69914
     HRTIMER:        510        302        368        260        219        255
 248        246
         RCU:      81290      68062      82979      69015      68390      69385       6
3304      63473
```

The `open_softirq` function takes two parameters:

- index of the interrupt;
- interrupt handler.

and adds interrupt handler to the array of the pending interrupts:

```
void open_softirq(int nr, void (*action)(struct softirq_action *))
{
        softirq_vec[nr].action = action;
}
```

In our case the interrupt handler is - `rcu_process_callbacks` which is defined in the kernel/rcu/tree.c and does the `RCU` core processing for the current CPU. After we registered `softirq` interrupt for the `RCU`, we can see the following code:

```
cpu_notifier(rcu_cpu_notify, 0);
pm_notifier(rcu_pm_notify, 0);
for_each_online_cpu(cpu)
    rcu_cpu_notify(NULL, CPU_UP_PREPARE, (void *)(long)cpu);
```

Here we can see registration of the `cpu` notifier which needs in systems which supports CPU hotplug and we will not dive into details about this theme. The last function in the `rcu_init` is the `rcu_early_boot_tests`:

```
void rcu_early_boot_tests(void)
{
        pr_info("Running RCU self tests\n");

        if (rcu_self_test)
                early_boot_test_call_rcu();
         if (rcu_self_test_bh)
                early_boot_test_call_rcu_bh();
         if (rcu_self_test_sched)
               early_boot_test_call_rcu_sched();
}
```

which runs self tests for the `RCU`.

That's all. We saw initialization process of the `RCU` subsystem. As I wrote above, more about the `RCU` will be in the separate chapter about synchronization primitives.

Rest of the initialization process

Ok, we already passed the main theme of this part which is `RCU` initialization, but it is not the end of the linux kernel initialization process. In the last paragraph of this theme we will see a couple of functions which work in the initialization time, but we will not dive into deep details around this function for different reasons. Some reasons not to dive into details are following:

- They are not very important for the generic kernel initialization process and depend on the different kernel configuration;
- They have the character of debugging and not important for now;
- We will see many of this stuff in the separate parts/chapters.

After we initialized `RCU`, the next step which you can see in the init/main.c is the - `trace_init` function. As you can understand from its name, this function initialize tracing subsystem. You can read more about linux kernel trace system - here.

After the `trace_init`, we can see the call of the `radix_tree_init`. If you are familiar with the different data structures, you can understand from the name of this function that it initializes kernel implementation of the Radix tree. This function is defined in the lib/radix-tree.c and you can read more about it in the part about Radix tree.

In the next step we can see the functions which are related to the `interrupts handling` subsystem, they are:

- `early_irq_init`
- `init_IRQ`
- `softirq_init`

We will see explanation about this functions and their implementation in the special part about interrupts and exceptions handling. After this many different functions (like `init_timers`, `hrtimers_init`, `time_init`, etc.) which are related to different timing and timers stuff. We will see more about these function in the chapter about timers.

The next couple of functions are related with the perf events - `perf_event-init` (there will be separate chapter about perf), initialization of the `profiling` with the `profile_init`. After this we enable `irq` with the call of the:

```
local_irq_enable();
```

which expands to the `sti` instruction and making post initialization of the SLAB with the call of the `kmem_cache_init_late` function (As I wrote above we will know about the `SLAB` in the Linux memory management chapter).

After the post initialization of the `SLAB`, next point is initialization of the console with the `console_init` function from the drivers/tty/tty_io.c.

After the console initialization, we can see the `lockdep_info` function which prints information about the Lock dependency validator. After this, we can see the initialization of the dynamic allocation of the `debug objects` with the `debug_objects_mem_init`, kernel memory leak detector initialization with the `kmemleak_init`, `percpu` pageset setup with the `setup_per_cpu_pageset`, setup of the NUMA policy with the `numa_policy_init`, setting time for the scheduler with the `sched_clock_init`, `pidmap` initialization with the call of the `pidmap_init` function for the initial `PID` namespace, cache creation with the `anon_vma_init` for the private virtual memory areas and early initialization of the ACPI with the `acpi_early_init`.

This is the end of the ninth part of the linux kernel initialization process and here we saw initialization of the RCU. In the last paragraph of this part (`Rest of the initialization process`) we will go through many functions but did not dive into details about their implementations. Do not worry if you do not know anything about these stuff or you know and do not understand anything about this. As I already wrote many times, we will see details of implementations in other parts or other chapters.

Conclusion

It is the end of the ninth part about the linux kernel initialization process. In this part, we looked on the initialization process of the `RCU` subsystem. In the next part we will continue to dive into linux kernel initialization process and I hope that we will finish with the `start_kernel` function and will go to the `rest_init` function from the same init/main.c source code file and will see the start of the first process.

If you have any questions or suggestions write me a comment or ping me at twitter.

Please note that English is not my first language, And I am really sorry for any inconvenience. If you find any mistakes please send me PR to linux-insides.

Links

- lock-free data structures
- kmemleak
- ACPI
- IRQs
- RCU
- RCU documentation
- integer ID management
- Documentation/memory-barriers.txt
- Runtime locking correctness validator
- Per-CPU variables
- Linux kernel memory management
- slab
- i2c
- Previous part

Kernel initialization. Part 10.

End of the linux kernel initialization process

This is tenth part of the chapter about linux kernel initialization process and in the previous part we saw the initialization of the RCU and stopped on the call of the `acpi_early_init` function. This part will be the last part of the Kernel initialization process chapter, so let's finish it.

After the call of the `acpi_early_init` function from the init/main.c, we can see the following code:

```
#ifdef CONFIG_X86_ESPFIX64
    init_espfix_bsp();
#endif
```

Here we can see the call of the `init_espfix_bsp` function which depends on the `CONFIG_X86_ESPFIX64` kernel configuration option. As we can understand from the function name, it does something with the stack. This function is defined in the arch/x86/kernel/espfix_64.c and prevents leaking of `31:16` bits of the `esp` register during returning to 16-bit stack. First of all we install `espfix` page upper directory into the kernel page directory in the `init_espfix_bs`:

```
pgd_p = &init_level4_pgt[pgd_index(ESPFIX_BASE_ADDR)];
pgd_populate(&init_mm, pgd_p, (pud_t *)espfix_pud_page);
```

Where `ESPFIX_BASE_ADDR` is:

```
#define PGDIR_SHIFT     39
#define ESPFIX_PGD_ENTRY _AC(-2, UL)
#define ESPFIX_BASE_ADDR (ESPFIX_PGD_ENTRY << PGDIR_SHIFT)
```

Also we can find it in the Documentation/x86/x86_64/mm:

```
... unused hole ...
ffffff0000000000 - ffffff7fffffffff (=39 bits) %esp fixup stacks
... unused hole ...
```

After we've filled page global directory with the `espfix` pud, the next step is call of the `init_espfix_random` and `init_espfix_ap` functions. The first function returns random locations for the `espfix` page and the second enables the `espfix` for the current CPU. After the `init_espfix_bsp` finished the work, we can see the call of the `thread_info_cache_init` function which defined in the kernel/fork.c and allocates cache for the `thread_info` if `THREAD_SIZE` is less than `PAGE_SIZE` :

```
# if THREAD_SIZE >= PAGE_SIZE
...
...
...
void thread_info_cache_init(void)
{
        thread_info_cache = kmem_cache_create("thread_info", THREAD_SIZE,
                                              THREAD_SIZE, 0, NULL);
        BUG_ON(thread_info_cache == NULL);
}
...
...
...
#endif
```

As we already know the `PAGE_SIZE` is `(_AC(1,UL) << PAGE_SHIFT)` or `4096` bytes and `THREAD_SIZE` is `(PAGE_SIZE << THREAD_SIZE_ORDER)` or `16384` bytes for the `x86_64` . The next function after the `thread_info_cache_init` is the `cred_init` from the kernel/cred.c. This function just allocates cache for the credentials (like `uid` , `gid` , etc.):

```
void __init cred_init(void)
{
         cred_jar = kmem_cache_create("cred_jar", sizeof(struct cred),
                                      0, SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL);
}
```

more about credentials you can read in the Documentation/security/credentials.txt. Next step is the `fork_init` function from the kernel/fork.c. The `fork_init` function allocates cache for the `task_struct` . Let's look on the implementation of the `fork_init` . First of all we can see definitions of the `ARCH_MIN_TASKALIGN` macro and creation of a slab where task_structs will be allocated:

```
#ifndef CONFIG_ARCH_TASK_STRUCT_ALLOCATOR
#ifndef ARCH_MIN_TASKALIGN
#define ARCH_MIN_TASKALIGN      L1_CACHE_BYTES
#endif
        task_struct_cachep =
                kmem_cache_create("task_struct", sizeof(struct task_struct),
                        ARCH_MIN_TASKALIGN, SLAB_PANIC | SLAB_NOTRACK, NULL);
#endif
```

As we can see this code depends on the `CONFIG_ARCH_TASK_STRUCT_ACLLOCATOR` kernel configuration option. This configuration option shows the presence of the `alloc_task_struct` for the given architecture. As `x86_64` has no `alloc_task_struct` function, this code will not work and even will not be compiled on the `x86_64`.

Allocating cache for init task

After this we can see the call of the `arch_task_cache_init` function in the `fork_init`:

```
void arch_task_cache_init(void)
{
        task_xstate_cachep =
                kmem_cache_create("task_xstate", xstate_size,
                                  __alignof__(union thread_xstate),
                                  SLAB_PANIC | SLAB_NOTRACK, NULL);
        setup_xstate_comp();
}
```

The `arch_task_cache_init` does initialization of the architecture-specific caches. In our case it is `x86_64`, so as we can see, the `arch_task_cache_init` allocates cache for the `task_xstate` which represents FPU state and sets up offsets and sizes of all extended states in xsave area with the call of the `setup_xstate_comp` function. After the `arch_task_cache_init` we calculate default maximum number of threads with the:

```
set_max_threads(MAX_THREADS);
```

where default maximum number of threads is:

```
#define FUTEX_TID_MASK  0x3fffffff
#define MAX_THREADS     FUTEX_TID_MASK
```

In the end of the `fork_init` function we initialize signal handler:

```
init_task.signal->rlim[RLIMIT_NPROC].rlim_cur = max_threads/2;
init_task.signal->rlim[RLIMIT_NPROC].rlim_max = max_threads/2;
init_task.signal->rlim[RLIMIT_SIGPENDING] =
        init_task.signal->rlim[RLIMIT_NPROC];
```

As we know the `init_task` is an instance of the `task_struct` structure, so it contains `signal` field which represents signal handler. It has following type `struct signal_struct`. On the first two lines we can see setting of the current and maximum limit of the `resource limits`. Every process has an associated set of resource limits. These limits specify amount of resources which current process can use. Here `rlim` is resource control limit and presented by the:

```
struct rlimit {
        __kernel_ulong_t        rlim_cur;
        __kernel_ulong_t        rlim_max;
};
```

structure from the include/uapi/linux/resource.h. In our case the resource is the `RLIMIT_NPROC` which is the maximum number of processes that user can own and `RLIMIT_SIGPENDING` - the maximum number of pending signals. We can see it in the:

```
cat /proc/self/limits
Limit                     Soft Limit           Hard Limit           Units
...
...
...
Max processes             63815                63815                processes
Max pending signals       63815                63815                signals
...
...
...
```

Initialization of the caches

The next function after the `fork_init` is the `proc_caches_init` from the kernel/fork.c. This function allocates caches for the memory descriptors (or `mm_struct` structure). At the beginning of the `proc_caches_init` we can see allocation of the different SLAB caches with the call of the `kmem_cache_create`:

- `sighand_cachep` - manage information about installed signal handlers;
- `signal_cachep` - manage information about process signal descriptor;
- `files_cachep` - manage information about opened files;

- `fs_cachep` - manage filesystem information.

After this we allocate `SLAB` cache for the `mm_struct` structures:

```
mm_cachep = kmem_cache_create("mm_struct",
                        sizeof(struct mm_struct), ARCH_MIN_MMSTRUCT_ALIGN,
                        SLAB_HWCACHE_ALIGN|SLAB_PANIC|SLAB_NOTRACK, NULL);
```

After this we allocate `SLAB` cache for the important `vm_area_struct` which used by the kernel to manage virtual memory space:

```
vm_area_cachep = KMEM_CACHE(vm_area_struct, SLAB_PANIC);
```

Note, that we use `KMEM_CACHE` macro here instead of the `kmem_cache_create`. This macro is defined in the include/linux/slab.h and just expands to the `kmem_cache_create` call:

```
#define KMEM_CACHE(__struct, __flags) kmem_cache_create(#__struct,\
                sizeof(struct __struct), __alignof__(struct __struct),\
                (__flags), NULL)
```

The `KMEM_CACHE` has one difference from `kmem_cache_create`. Take a look on `__alignof__` operator. The `KMEM_CACHE` macro aligns `SLAB` to the size of the given structure, but `kmem_cache_create` uses given value to align space. After this we can see the call of the `mmap_init` and `nsproxy_cache_init` functions. The first function initializes virtual memory area `SLAB` and the second function initializes `SLAB` for namespaces.

The next function after the `proc_caches_init` is `buffer_init`. This function is defined in the fs/buffer.c source code file and allocate cache for the `buffer_head`. The `buffer_head` is a special structure which defined in the include/linux/buffer_head.h and used for managing buffers. In the start of the `buffer_init` function we allocate cache for the `struct buffer_head` structures with the call of the `kmem_cache_create` function as we did in the previous functions. And calculate the maximum size of the buffers in memory with:

```
nrpages = (nr_free_buffer_pages() * 10) / 100;
max_buffer_heads = nrpages * (PAGE_SIZE / sizeof(struct buffer_head));
```

which will be equal to the `10%` of the `ZONE_NORMAL` (all RAM from the 4GB on the `x86_64`). The next function after the `buffer_init` is - `vfs_caches_init`. This function allocates `SLAB` caches and hashtable for different VFS caches. We already saw the `vfs_caches_init_early` function in the eighth part of the linux kernel initialization process which initialized caches for `dcache` (or directory-cache) and inode cache. The `vfs_caches_init` function makes post-early initialization of the `dcache` and `inode` caches, private data cache, hash tables for the

mount points, etc. More details about VFS will be described in the separate part. After this we can see `signals_init` function. This function is defined in the kernel/signal.c and allocates a cache for the `sigqueue` structures which represents queue of the real time signals. The next function is `page_writeback_init`. This function initializes the ratio for the dirty pages. Every low-level page entry contains the `dirty` bit which indicates whether a page has been written to after been loaded into memory.

Creation of the root for the procfs

After all of this preparations we need to create the root for the proc filesystem. We will do it with the call of the `proc_root_init` function from the fs/proc/root.c. At the start of the `proc_root_init` function we allocate the cache for the inodes and register a new filesystem in the system with the:

```
err = register_filesystem(&proc_fs_type);
      if (err)
              return;
```

As I wrote above we will not dive into details about VFS and different filesystems in this chapter, but will see it in the chapter about the `VFS`. After we've registered a new filesystem in our system, we call the `proc_self_init` function from the fs/proc/self.c and this function allocates `inode` number for the `self` (`/proc/self` directory refers to the process accessing the `/proc` filesystem). The next step after the `proc_self_init` is `proc_setup_thread_self` which setups the `/proc/thread-self` directory which contains information about current thread. After this we create `/proc/self/mounts` symlink which will contains mount points with the call of the

```
proc_symlink("mounts", NULL, "self/mounts");
```

and a couple of directories depends on the different configuration options:

```
#ifdef CONFIG_SYSVIPC
        proc_mkdir("sysvipc", NULL);
#endif
        proc_mkdir("fs", NULL);
        proc_mkdir("driver", NULL);
        proc_mkdir("fs/nfsd", NULL);
#if defined(CONFIG_SUN_OPENPROMFS) || defined(CONFIG_SUN_OPENPROMFS_MODULE)
        proc_mkdir("openprom", NULL);
#endif
        proc_mkdir("bus", NULL);
        ...
        ...
        ...
        if (!proc_mkdir("tty", NULL))
                 return;
        proc_mkdir("tty/ldisc", NULL);
        ...
        ...
        ...
```

In the end of the `proc_root_init` we call the `proc_sys_init` function which creates `/proc/sys` directory and initializes the Sysctl.

It is the end of `start_kernel` function. I did not describe all functions which are called in the `start_kernel`. I skipped them, because they are not important for the generic kernel initialization stuff and depend on only different kernel configurations. They are `taskstats_init_early` which exports per-task statistic to the user-space, `delayacct_init` - initializes per-task delay accounting, `key_init` and `security_init` initialize different security stuff, `check_bugs` - fix some architecture-dependent bugs, `ftrace_init` function executes initialization of the ftrace, `cgroup_init` makes initialization of the rest of the cgroup subsystem,etc. Many of these parts and subsystems will be described in the other chapters.

That's all. Finally we have passed through the long-long `start_kernel` function. But it is not the end of the linux kernel initialization process. We haven't run the first process yet. In the end of the `start_kernel` we can see the last call of the - `rest_init` function. Let's go ahead.

First steps after the start_kernel

The `rest_init` function is defined in the same source code file as `start_kernel` function, and this file is init/main.c. In the beginning of the `rest_init` we can see call of the two following functions:

```
    rcu_scheduler_starting();
    smpboot_thread_init();
```

The first `rcu_scheduler_starting` makes RCU scheduler active and the second `smpboot_thread_init` registers the `smpboot_thread_notifier` CPU notifier (more about it you can read in the CPU hotplug documentation. After this we can see the following calls:

```
kernel_thread(kernel_init, NULL, CLONE_FS);
pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);
```

Here the `kernel_thread` function (defined in the kernel/fork.c) creates new kernel thread.As we can see the `kernel_thread` function takes three arguments:

- Function which will be executed in a new thread;
- Parameter for the `kernel_init` function;
- Flags.

We will not dive into details about `kernel_thread` implementation (we will see it in the chapter which describe scheduler, just need to say that `kernel_thread` invokes clone). Now we only need to know that we create new kernel thread with `kernel_thread` function, parent and child of the thread will use shared information about filesystem and it will start to execute `kernel_init` function. A kernel thread differs from a user thread that it runs in kernel mode. So with these two `kernel_thread` calls we create two new kernel threads with the `PID = 1` for `init` process and `PID = 2` for `kthreadd`. We already know what is `init` process. Let's look on the `kthreadd`. It is a special kernel thread which manages and helps different parts of the kernel to create another kernel thread. We can see it in the output of the `ps` util:

```
$ ps -ef | grep kthread
root         2     0  0 Jan11 ?        00:00:00 [kthreadd]
```

Let's postpone `kernel_init` and `kthreadd` for now and go ahead in the `rest_init`. In the next step after we have created two new kernel threads we can see the following code:

```
    rcu_read_lock();
    kthreadd_task = find_task_by_pid_ns(pid, &init_pid_ns);
    rcu_read_unlock();
```

The first `rcu_read_lock` function marks the beginning of an RCU read-side critical section and the `rcu_read_unlock` marks the end of an RCU read-side critical section. We call these functions because we need to protect the `find_task_by_pid_ns`. The `find_task_by_pid_ns`

returns pointer to the `task_struct` by the given pid. So, here we are getting the pointer to the `task_struct` for `PID = 2` (we got it after `kthreadd` creation with the `kernel_thread`). In the next step we call `complete` function

```
complete(&kthreadd_done);
```

and pass address of the `kthreadd_done`. The `kthreadd_done` defined as

```
static __initdata DECLARE_COMPLETION(kthreadd_done);
```

where `DECLARE_COMPLETION` macro defined as:

```
#define DECLARE_COMPLETION(work) \
         struct completion work = COMPLETION_INITIALIZER(work)
```

and expands to the definition of the `completion` structure. This structure is defined in the include/linux/completion.h and presents `completions` concept. Completions is a code synchronization mechanism which provides race-free solution for the threads that must wait for some process to have reached a point or a specific state. Using completions consists of three parts: The first is definition of the `complete` structure and we did it with the `DECLARE_COMPLETION`. The second is call of the `wait_for_completion`. After the call of this function, a thread which called it will not continue to execute and will wait while other thread did not call `complete` function. Note that we call `wait_for_completion` with the `kthreadd_done` in the beginning of the `kernel_init_freeable`:

```
wait_for_completion(&kthreadd_done);
```

And the last step is to call `complete` function as we saw it above. After this the `kernel_init_freeable` function will not be executed while `kthreadd` thread will not be set. After the `kthreadd` was set, we can see three following functions in the `rest_init`:

```
    init_idle_bootup_task(current);
    schedule_preempt_disabled();
    cpu_startup_entry(CPUHP_ONLINE);
```

The first `init_idle_bootup_task` function from the kernel/sched/core.c sets the Scheduling class for the current process (`idle` class in our case):

```
void init_idle_bootup_task(struct task_struct *idle)
{
        idle->sched_class = &idle_sched_class;
}
```

where `idle` class is a low task priority and tasks can be run only when the processor doesn't have anything to run besides this tasks. The second function `schedule_preempt_disabled` disables preempt in `idle` tasks. And the third function `cpu_startup_entry` is defined in the kernel/sched/idle.c and calls `cpu_idle_loop` from the kernel/sched/idle.c. The `cpu_idle_loop` function works as process with `PID = 0` and works in the background. Main purpose of the `cpu_idle_loop` is to consume the idle CPU cycles. When there is no process to run, this process starts to work. We have one process with `idle` scheduling class (we just set the `current` task to the `idle` with the call of the `init_idle_bootup_task` function), so the `idle` thread does not do useful work but just checks if there is an active task to switch to:

```
static void cpu_idle_loop(void)
{
        ...
        ...
        ...
        while (1) {
                while (!need_resched()) {
                ...
                ...
                ...
                }
        ...
        }
```

More about it will be in the chapter about scheduler. So for this moment the `start_kernel` calls the `rest_init` function which spawns an `init` (`kernel_init` function) process and become `idle` process itself. Now is time to look on the `kernel_init`. Execution of the `kernel_init` function starts from the call of the `kernel_init_freeable` function. The `kernel_init_freeable` function first of all waits for the completion of the `kthreadd` setup. I already wrote about it above:

```
wait_for_completion(&kthreadd_done);
```

After this we set `gfp_allowed_mask` to `__GFP_BITS_MASK` which means that system is already running, set allowed cpus/mems to all CPUs and NUMA nodes with the `set_mems_allowed` function, allow `init` process to run on any CPU with the `set_cpus_allowed_ptr`, set pid for the `cad` or `Ctrl-Alt-Delete`, do preparation for booting of the other CPUs with the call of

the `smp_prepare_cpus`, call early initcalls with the `do_pre_smp_initcalls`, initialize `SMP` with the `smp_init` and initialize lockup_detector with the call of the `lockup_detector_init` and initialize scheduler with the `sched_init_smp`.

After this we can see the call of the following functions - `do_basic_setup`. Before we will call the `do_basic_setup` function, our kernel already initialized for this moment. As comment says:

```
Now we can finally start doing some real work..
```

The `do_basic_setup` will reinitialize cpuset to the active CPUs, initialize the `khelper` - which is a kernel thread which used for making calls out to userspace from within the kernel, initialize tmpfs, initialize `drivers` subsystem, enable the user-mode helper `workqueue` and make post-early call of the `initcalls`. We can see opening of the `dev/console` and dup twice file descriptors from `0` to `2` after the `do_basic_setup`:

```
if (sys_open((const char __user *) "/dev/console", O_RDWR, 0) < 0)
    pr_err("Warning: unable to open an initial console.\n");

(void) sys_dup(0);
(void) sys_dup(0);
```

We are using two system calls here `sys_open` and `sys_dup`. In the next chapters we will see explanation and implementation of the different system calls. After we opened initial console, we check that `rdinit=` option was passed to the kernel command line or set default path of the ramdisk:

```
if (!ramdisk_execute_command)
    ramdisk_execute_command = "/init";
```

Check user's permissions for the `ramdisk` and call the `prepare_namespace` function from the init/do_mounts.c which checks and mounts the initrd:

```
if (sys_access((const char __user *) ramdisk_execute_command, 0) != 0) {
    ramdisk_execute_command = NULL;
    prepare_namespace();
}
```

This is the end of the `kernel_init_freeable` function and we need return to the `kernel_init`. The next step after the `kernel_init_freeable` finished its execution is the `async_synchronize_full`. This function waits until all asynchronous function calls have been done and after it we will call the `free_initmem` which will release all memory occupied by the

initialization stuff which located between `__init_begin` and `__init_end`. After this we protect `.rodata` with the `mark_rodata_ro` and update state of the system from the `SYSTEM_BOOTING` to the

```
system_state = SYSTEM_RUNNING;
```

And tries to run the `init` process:

```
if (ramdisk_execute_command) {
    ret = run_init_process(ramdisk_execute_command);
    if (!ret)
        return 0;
    pr_err("Failed to execute %s (error %d)\n",
           ramdisk_execute_command, ret);
}
```

First of all it checks the `ramdisk_execute_command` which we set in the `kernel_init_freeable` function and it will be equal to the value of the `rdinit=` kernel command line parameters or `/init` by default. The `run_init_process` function fills the first element of the `argv_init` array:

```
static const char *argv_init[MAX_INIT_ARGS+2] = { "init", NULL, };
```

which represents arguments of the `init` program and call `do_execve` function:

```
argv_init[0] = init_filename;
return do_execve(getname_kernel(init_filename),
    (const char __user *const __user *)argv_init,
    (const char __user *const __user *)envp_init);
```

The `do_execve` function is defined in the include/linux/sched.h and runs program with the given file name and arguments. If we did not pass `rdinit=` option to the kernel command line, kernel starts to check the `execute_command` which is equal to value of the `init=` kernel command line parameter:

```
    if (execute_command) {
        ret = run_init_process(execute_command);
        if (!ret)
            return 0;
        panic("Requested init %s failed (error %d).",
              execute_command, ret);
    }
```

If we did not pass `init=` kernel command line parameter either, kernel tries to run one of the following executable files:

```
if (!try_to_run_init_process("/sbin/init") ||
    !try_to_run_init_process("/etc/init") ||
    !try_to_run_init_process("/bin/init") ||
    !try_to_run_init_process("/bin/sh"))
    return 0;
```

Otherwise we finish with panic:

```
panic("No working init found.  Try passing init= option to kernel. "
      "See Linux Documentation/init.txt for guidance.");
```

That's all! Linux kernel initialization process is finished!

Conclusion

It is the end of the tenth part about the linux kernel initialization process. It is not only the `tenth` part, but also is the last part which describes initialization of the linux kernel. As I wrote in the first part of this chapter, we will go through all steps of the kernel initialization and we did it. We started at the first architecture-independent function - `start_kernel` and finished with the launch of the first `init` process in the our system. I skipped details about different subsystem of the kernel, for example I almost did not cover scheduler, interrupts, exception handling, etc. From the next part we will start to dive to the different kernel subsystems. Hope it will be interesting.

If you have any questions or suggestions write me a comment or ping me at twitter.

Please note that English is not my first language, And I am really sorry for any inconvenience. If you find any mistakes please send me PR to linux-insides.

Links

- SLAB
- xsave
- FPU
- Documentation/security/credentials.txt
- Documentation/x86/x86_64/mm
- RCU
- VFS

- inode
- proc
- man proc
- Sysctl
- ftrace
- cgroup
- CPU hotplug documentation
- completions - wait for completion handling
- NUMA
- cpus/mems
- initcalls
- Tmpfs
- initrd
- panic
- Previous part

中断和中断处理

在 linux 内核中你会发现很多关于中断和异常处理的话题

- 中断和中断处理第一部分 - 描述中断处理主题
- 深入 Linux 内核中的中断 - 这部分开始描述和初步步骤相关的中断和异常处理。
- 初步中断处理 - 描述初步中断处理。
- 中断处理 - fourth part describes first non-early interrupt handlers.
- 异常处理的实现 - 一些异常处理的实现，比如双重错误、除零等等。
- 处理不可屏蔽中断 - 描述了如何处理不可屏蔽的中断和剩下的一些与特定架构相关的中断。
- 深入外部硬件中断 - 这部分讲述了关于处理外部硬件中断的一些早期初始化代码。
- IRQs的非早期初始化 - 这部分讲述了处理外部硬件中断的非早期初始化代码。
- Softirq, Tasklets and Workqueues - 这部分讲述了softirqs、tasklets 和 workqueues 的内容。
- 最后一部分 - 这是中断和中断处理的最后一部分，并且我们将会看到一个真实的硬件驱动和中断。

中断和中断处理 Part 1.

Introduction

这是 linux 内核揭密 这本书最新章节的第一部分。我们已经在这本书前面的章节中走过了漫长的道路。从内核初始化的第一步开始，结束于第一个 `init` 程序的启动。我们见证了一系列与各种内核子系统相关的初始化步骤，但是我们并没有深入这些子系统。在这一章中，我们将会试着去了解这些内核子系统是如何工作和实现的。就像你在这章标题中看到的，第一个子系统是中断（interrupts）。

什么是中断？

我们已经在这本书的很多地方听到过 `中断（interrupts）` 这个词，也看到过很多关于中断的例子。在这一章中我们将会从下面的主题开始：

- 什么是 `中断（interrupts）` ？
- 什么是 `中断处理（interrupt handlers）` ？

我们将会继续深入探讨 `中断` 的细节和 Linux 内核如何处理这些中断。

所以，首先什么是中断？中断就是当软件或者硬件需要使用 CPU 时引发的 `事件（event）` 。比如，当我们在键盘上按下一个键的时候，我们下一步期望做什么？操作系统和电脑应该怎么做？做一个简单的假设，每一个物理硬件都有一根连接 CPU 的中断线，设备可以通过它对 CPU 发起中断信号。但是中断信号并不是直接发送给 CPU。在老机器上中断信号发送给 PIC，它是一个顺序处理各种设备的各种中断请求的芯片。在新机器上，则是高级程序中断控制器（Advanced Programmable Interrupt Controller）做这件事情，即我们熟知的 `APIC` 。一个 APIC 包括两个独立的设备：

- `Local APIC`
- `I/O APIC`

第一个设备 - `Local APIC` 存在于每个CPU核心中，Local APIC 负责处理特定于 CPU 的中断配置。Local APIC 常被用于管理来自 APIC 时钟（APIC-timer）、热敏元件和其他与 I/O 设备连接的设备的中断。

第二个设备 - `I/O APIC` 提供了多核处理器的中断管理。它被用来在所有的 CPU 核心中分发外部中断。更多关于 local 和 I/O APIC 的内容将会在这一节的下面讲到。就如你所知道的，中断可以在任何时间发生。当一个中断发生时，操作系统必须立刻处理它。但是 `处理一个中断` 是什么意思呢？当一个中断发生时，操作系统必须确保下面的步骤顺序：

- 内核必须暂停执行当前进程(取代当前的任务)；
- 内核必须搜索中断处理程序并且转交控制权(执行中断处理程序)；
- 中断处理程序结束之后，被中断的进程能够恢复执行。

当然，在这个中断处理程序中会涉及到很多错综复杂的过程。但是上面 3 条是这个程序的基本骨架。

每个中断处理程序的地址都保存在一个特殊的位置，这个位置被称为 `中断描述符表（Interrupt Descriptor Table）` 或者 `IDT` 。处理器使用一个唯一的数字来识别中断和异常的类型，这个数字被称为 `中断标识码（vector number）` 。一个中断标识码就是一个 `IDT` 的标识。中断标识码范围是有限的，从 `0` 到 `255` 。你可以在 Linux 内核源码中找到下面的中断标识码范围检查代码：

```
BUG_ON((unsigned)n > 0xFF);
```

你可以在 Linux 内核源码中关于中断设置的地方找到这个检查(例如：`set_intr_gate`，`void set_system_intr_gate` 在 arch/x86/include/asm/desc.h中)。从 `0` 到 `31` 的 32 个中断标识码被处理器保留，用作处理架构定义的异常和中断。你可以在 Linux 内核初始化程序的第二部分 - 早期中断和异常处理中找到这个表和关于这些中断标识码的描述。从 `32` 到 `255` 的中断标识码设计为用户定义中断并且不被系统保留。这些中断通常分配给外部 I/O 设备，使这些设备可以发送中断给处理器。

现在，我们来讨论中断的类型。笼统地来讲，我们可以把中断分为两个主要类型：

- 外部或者硬件引起的中断；
- 软件引起的中断。

第一种类型 - 外部中断，由 `Local APIC` 或者与 `Local APIC` 连接的处理器针脚接收。第二种类型 - 软件引起的中断，由处理器自身的特殊情况引起(有时使用特殊架构的指令)。一个常见的关于特殊情况的例子就是 `除零` 。另一个例子就是使用 `系统调用（syscall）` 退出程序。

就如之前提到过的，中断可以在任何时间因为超出代码和 CPU 控制的原因而发生。另一方面，异常和程序执行 `同步（synchronous）` ，并且可以被分为 3 类：

- `故障（Faults）`
- `陷入（Traps）`
- `终止（Aborts）`

`故障` 是在执行一个“不完善的”指令（可以在之后被修正）之前被报告的异常。如果发生了，它允许被中断的程序继续执行。

接下来的 `陷入` 是一个在执行了 `陷入` 指令后立刻被报告的异常。陷入同样允许被中断的程序继续执行，就像 `故障` 一样。

最后的 `终止` 是一个从不报告引起异常的精确指令的异常，并且不允许被中断的程序继续执行。

我们已经从前面的部分知道，中断可以分为 `可屏蔽的（maskable）` 和 `不可屏蔽的（non-maskable）` 。可屏蔽的中断可以被阻塞，使用 `x86_64` 的指令 - `sti` 和 `cli` 。我们可以在 Linux 内核代码中找到他们：

```
static inline void native_irq_disable(void)
{
        asm volatile("cli": : :"memory");
}
```

and

```
static inline void native_irq_enable(void)
{
        asm volatile("sti": : :"memory");
}
```

这两个指令修改了在中断寄存器中的 `IF` 标识位。 `sti` 指令设置 `IF` 标识， `cli` 指令清除这个标识。不可屏蔽的中断总是被报告。通常，任何硬件上的失败都映射为不可屏蔽中断。

如果多个异常或者中断同时发生，处理器以事先设定好的中断优先级处理他们。我们可以定义下面表中的从最低到最高的优先级：

```
+----------------------------------------------------------------+
|              |                                                  |
|   Priority   | Description                                      |
|              |                                                  |
+--------------+--------------------------------------------------+
|              | Hardware Reset and Machine Checks                |
|     1        | - RESET                                          |
|              | - Machine Check                                  |
+--------------+--------------------------------------------------+
|              | Trap on Task Switch                              |
|     2        | - T flag in TSS is set                           |
|              |                                                  |
+--------------+--------------------------------------------------+
|              | External Hardware Interventions                  |
|              | - FLUSH                                          |
|     3        | - STOPCLK                                        |
|              | - SMI                                            |
|              | - INIT                                           |
+--------------+--------------------------------------------------+
|              | Traps on the Previous Instruction                |
|     4        | - Breakpoints                                    |
```

```
|              | - Debug Trap Exceptions                        |
+--------------+------------------------------------------------+
|     5        | Nonmaskable Interrupts                         |
+--------------+------------------------------------------------+
|     6        | Maskable Hardware Interrupts                   |
+--------------+------------------------------------------------+
|     7        | Code Breakpoint Fault                          |
+--------------+------------------------------------------------+
|     8        | Faults from Fetching Next Instruction          |
|              | Code-Segment Limit Violation                   |
|              | Code Page Fault                                |
+--------------+------------------------------------------------+
|              | Faults from Decoding the Next Instruction      |
|              | Instruction length > 15 bytes                  |
|     9        | Invalid Opcode                                 |
|              | Coprocessor Not Available                      |
|              |                                                |
+--------------+------------------------------------------------+
|     10       | Faults on Executing an Instruction             |
|              | Overflow                                       |
|              | Bound error                                    |
|              | Invalid TSS                                    |
|              | Segment Not Present                            |
|              | Stack fault                                    |
|              | General Protection                             |
|              | Data Page Fault                                |
|              | Alignment Check                                |
|              | x87 FPU Floating-point exception               |
|              | SIMD floating-point exception                  |
|              | Virtualization exception                       |
+--------------+------------------------------------------------+
```

现在我们了解了一些关于各种类型的中断和异常的内容，是时候转到更实用的部分了。我们从 `中断描述符表（IDT）` 开始。就如之前所提到的，`IDT` 保存了中断和异常处理程序的入口指针。`IDT` 是一个类似于 `全局描述符表（Global Descriptor Table）` 的结构，我们在内核启动程序的第二部分已经介绍过。但是他们确实有一些不同，`IDT` 的表项被称为 `门（gates）`，而不是 `描述符（descriptors）`。它可以包含下面的一种：

- 中断门（Interrupt gates）
- 任务门（Task gates）
- 陷阱门（Trap gates）

在 `x86` 架构中，只有 long mode 中断门和陷阱门可以在 `x86_64` 中引用。就像 `全局描述符表`，`中断描述符表` 在 `x86` 上是一个 8 字节数组门，而在 `x86_64` 上是一个 16 字节数组门。让我们回忆在内核启动程序的第二部分，`全局描述符表` 必须包含 `NULL` 描述符作为它的第一个元素。与 `全局描述符表` 不一样的是，`中断描述符表` 的第一个元素可以是一个门。它并不是强制要求的。比如，你可能还记得我们只是在早期的章节中过渡到保护模式时用 `NULL` 门加载过中断描述符表：

```
/*
 * Set up the IDT
 */
static void setup_idt(void)
{
	static const struct gdt_ptr null_idt = {0, 0};
	asm volatile("lidtl %0" : : "m" (null_idt));
}
```

在 arch/x86/boot/pm.c中。`中断描述符表` 可以在线性地址空间和基址的任何地方被加载，只要在 `x86` 上以 8 字节对齐，在 `x86_64` 上以 16 字节对齐。`IDT` 的基址存储在一个特殊的寄存器 - IDTR。在 `x86` 上有两个指令 - 协调工作来修改 `IDTR` 寄存器：

- `LIDT`
- `SIDT`

第一个指令 `LIDT` 用来加载 `IDT` 的基址，即在 `IDTR` 的指定操作数。第二个指令 `SIDT` 用来在指定操作数中读取和存储 `IDTR` 的内容。在 `x86` 上 `IDTR` 寄存器是 48 位，包含了下面的信息：

```
+-----------------------------------+----------------------+
|                                   |                      |
|     Base address of the IDT       |   Limit of the IDT   |
|                                   |                      |
+-----------------------------------+----------------------+
47                                16 15                    0
```

让我们看看 `setup_idt` 的实现，我们准备了一个 `null_idt`，并且使用 `lidt` 指令把它加载到 `IDTR` 寄存器。注意，`null_idt` 是 `gdt_ptr` 类型，后者定义如下：

```
struct gdt_ptr {
        u16 len;
        u32 ptr;
} __attribute__((packed));
```

这里我们可以看看 `IDTR` 结构的定义，就像我们在示意图中看到的一样，由 2 字节和 4 字节（共 48 位）的两个域组成。现在，让我们看看 `IDT` 入口结构体，它是一个在 `x86` 中被称为门的 16 字节数组。它拥有下面的结构：

```
127                                                                             96
+-------------------------------------------------------------------------------+
|                                                                               |
|                                Reserved                                       |
|                                                                               |
+--------------------------------------------------------------------------------
95                                                                              64
+-------------------------------------------------------------------------------+
|                                                                               |
|                               Offset 63..32                                   |
|                                                                               |
+-------------------------------------------------------------------------------+
63                               48 47      46  44   42    39             34    32
+-------------------------------------------------------------------------------+
|                                  |       |  D  |   |      |      |   |   |    |
|       Offset 31..16              |   P   |  P  | 0 |Type |0 0 0 | 0 | 0 | IST |
|                                  |       |  L  |   |      |      |   |   |    |
 -------------------------------------------------------------------------------+
31                                   16 15                                      0
+-------------------------------------------------------------------------------+
|                                      |                                        |
|          Segment Selector            |                 Offset 15..0           |
|                                      |                                        |
+-------------------------------------------------------------------------------+
```

为了把索引格式化成 IDT 的格式，处理器把异常和中断向量分为 16 个级别。处理器处理异常和中断的发生就像它看到 `call` 指令时处理一个程序调用一样。处理器使用中断或异常的唯一的数字或 `中断标识码` 作为索引来寻找对应的 `中断描述符表` 的条目。现在让我们更近距离地看看 `IDT` 条目。

就像我们所看到的一样，在表中的 `IDT` 条目由下面的域组成：

- `0-15` bits - 段选择器偏移，处理器用它作为中断处理程序的入口指针基址；
- `16-31` bits - 段选择器基址，包含中断处理程序入口指针；
- `IST` - 在 `x86_64` 上的一个新的机制，下面我们会介绍它；
- `DPL` - 描述符特权级；
- `P` - 段存在标志；
- `48-63` bits - 中断处理程序基址的第二部分；
- `64-95` bits - 中断处理程序基址的第三部分；
- `96-127` bits - CPU 保留位.

`Type` 域描述了 `IDT` 条目的类型。有三种不同的中断处理程序：

- 中断门（Interrupt gate）
- 陷入门（Trap gate）
- 任务门（Task gate）

`IST` 或者说是 `Interrupt Stack Table` 是 `x86_64` 中的新机制，它用来代替传统的栈切换机制。之前的 `x86` 架构提供的机制可以在响应中断时自动切换栈帧。 `IST` 是 `x86` 栈切换模式的一个修改版，在它使能之后可以无条件地切换栈，并且可以被任何与确定中断（我们将在下面介绍它）关联的 `IDT` 条目中的中断使能。从这里可以看出， `IST` 并不是所有的中断必须的，一些中断可以继续使用传统的栈切换模式。 `IST` 机制在任务状态段（Task State Segment）或者 `TSS` 中提供了 7 个 `IST` 指针。 `TSS` 是一个包含进程信息的特殊结构，用来在执行中断或者处理 Linux 内核异常的时候做栈切换。每一个指针都被 `IDT` 中的中断门引用。

`中断描述符表` 使用 `gate_desc` 的数组描述：

```
extern gate_desc idt_table[];
```

`gate_desc` 定义如下：

```
#ifdef CONFIG_X86_64
...
...
...
typedef struct gate_struct64 gate_desc;
...
...
...
#endif
```

`gate_struct64` 定义如下：

```
struct gate_struct64 {
        u16 offset_low;
        u16 segment;
        unsigned ist : 3, zero0 : 5, type : 5, dpl : 2, p : 1;
        u16 offset_middle;
        u32 offset_high;
        u32 zero1;
} __attribute__((packed));
```

在 `x86_64` 架构中，每一个活动的线程在 Linux 内核中都有一个很大的栈。这个栈的大小由 `THREAD_SIZE` 定义，而且与下面的定义相等：

```
#define PAGE_SHIFT      12
#define PAGE_SIZE       (_AC(1,UL) << PAGE_SHIFT)
...
...
...
#define THREAD_SIZE_ORDER       (2 + KASAN_STACK_ORDER)
#define THREAD_SIZE  (PAGE_SIZE << THREAD_SIZE_ORDER)
```

`PAGE_SIZE` 是 `4096` 字节，`THREAD_SIZE_ORDER` 的值依赖于 `KASAN_STACK_ORDER` 。就像我们看到的，`KASAN_STACK` 依赖于 `CONFIG_KASAN` 内核配置参数，它定义如下：

```
#ifdef CONFIG_KASAN
    #define KASAN_STACK_ORDER 1
#else
    #define KASAN_STACK_ORDER 0
#endif
```

`KASan` 是一个运行时内存调试器。所以，如果 `CONFIG_KASAN` 被禁用，`THREAD_SIZE` 是 `16384` ；如果内核配置选项打开，`THREAD_SIZE` 的值是 `32768` 。这块栈空间保存着有用的数据，只要线程是活动状态或者僵尸状态。但是当线程在用户空间的时候，这个内核栈是空的，除非 `thread_info` 结构（关于这个结构的详细信息在 Linux 内核初始程序的第四部分）在这个栈空间的底部。活动的或者僵尸线程并不是在他们栈中的唯一的线程，与每一个 CPU 关联的特殊栈也存在于这个空间。当内核在这个 CPU 上执行代码的时候，这些栈处于活动状态；当在这个 CPU 上执行用户空间代码时，这些栈不包含任何有用的信息。每一个 CPU 也有一个特殊的 per-cpu 栈。首先是给外部中断使用的 `中断栈（interrupt stack）` 。它的大小定义如下：

```
#define IRQ_STACK_ORDER (2 + KASAN_STACK_ORDER)
#define IRQ_STACK_SIZE (PAGE_SIZE << IRQ_STACK_ORDER)
```

或者是 `16384` 字节。Per-cpu 的中断栈在 `x86_64` 架构中使用 `irq_stack_union` 联合描述:

```
union irq_stack_union {
    char irq_stack[IRQ_STACK_SIZE];

    struct {
        char gs_base[40];
        unsigned long stack_canary;
    };
};
```

第一个 `irq_stack` 域是一个 16KB 的数组。然后你可以看到 `irq_stack_union` 联合包含了一个结构体，这个结构体有两个域：

* `gs_base` - 总是指向 `irqstack` 联合底部的 `gs` 寄存器。在 `x86_64` 中，per-cpu（更多关于 `per-cpu` 变量的信息可以阅读特定的章节） 和 stack canary 共享 `gs` 寄存器。所有的 per-cpu 标志初始值为零，并且 `gs` 指向 per-cpu 区域的开始。你已经知道段内存模式已经废除很长时间了，但是我们可以使用特殊模块寄存器（Model specific registers）给这两个段寄存器 - `fs` 和 `gs` 设置基址，并且这些寄存器仍然可以被用作地址寄存器。如果你记得 Linux 内核初始程序的第一部分，你会记起我们设置了 `gs` 寄存器：

```
    movl    $MSR_GS_BASE,%ecx
    movl    initial_gs(%rip),%eax
    movl    initial_gs+4(%rip),%edx
    wrmsr
```

`initial_gs` 指向 `irq_stack_union`：

```
GLOBAL(initial_gs)
.quad    INIT_PER_CPU_VAR(irq_stack_union)
```

* `stack_canary` - Stack canary 对于中断栈来说是一个用来验证栈是否已经被修改的 `栈保护者（stack protector）` 。`gs_base` 是一个 40 字节的数组，`GCC` 要求 stack canary 在被修正过的偏移量上，并且 `gs` 的值在 `x86_64` 架构上必须是 `40`，在 `x86` 架构上必须是 `20`。

`irq_stack_union` 是 `percpu` 的第一个数据, 我们可以在 `System.map` 中看到它：

```
0000000000000000 D __per_cpu_start
0000000000000000 D irq_stack_union
0000000000004000 d exception_stacks
0000000000009000 D gdt_page
...
...
...
```

我们可以看到它在代码中的定义:

```
DECLARE_PER_CPU_FIRST(union irq_stack_union, irq_stack_union) __visible;
```

现在，是时候来看 `irq_stack_union` 的初始化过程了。除了 `irq_stack_union` 的定义，我们可以在arch/x86/include/asm/processor.h中查看下面的 per-cpu 变量

```
DECLARE_PER_CPU(char *, irq_stack_ptr);
DECLARE_PER_CPU(unsigned int, irq_count);
```

第一个就是 `irq_stack_ptr` 。从这个变量的名字中可以知道，它显然是一个指向这个栈顶的指针。第二个 `irq_count` 用来检查 CPU 是否已经在中断栈。`irq_stack_ptr` 的初始化在arch/x86/kernel/setup_percpu.c的 `setup_per_cpu_areas` 函数中：

```
void __init setup_per_cpu_areas(void)
{
...
...
#ifdef CONFIG_X86_64
for_each_possible_cpu(cpu) {
    ...
    ...
    ...
    per_cpu(irq_stack_ptr, cpu) =
            per_cpu(irq_stack_union.irq_stack, cpu) +
            IRQ_STACK_SIZE - 64;
    ...
    ...
    ...
#endif
...
...
}
```

现在，我们一个一个查看所有 CPU，并且设置 `irq_stack_ptr` 。事实证明它等于中断栈的顶减去 `64` 。为什么是 `64` ？TODO [arch/x86/kernel/cpu/common.c] 代码如下：

```
void load_percpu_segment(int cpu)
{
        ...
        ...
        ...
        loadsegment(gs, 0);
        wrmsrl(MSR_GS_BASE, (unsigned long)per_cpu(irq_stack_union.gs_base, cpu));
}
```

就像我们所知道的一样，`gs` 寄存器指向中断栈的栈底：

```
    movl    $MSR_GS_BASE,%ecx
    movl    initial_gs(%rip),%eax
    movl    initial_gs+4(%rip),%edx
    wrmsr

    GLOBAL(initial_gs)
    .quad    INIT_PER_CPU_VAR(irq_stack_union)
```

现在我们可以看到 `wrmsr` 指令，这个指令从 `edx:eax` 加载数据到 被 `ecx` 指向的MSR寄存器)。在这里MSR寄存器是 `MSR_GS_BASE` ，它保存了被 `gs` 寄存器指向的内存段的基址。 `edx:eax` 指向 `initial_gs` 的地址，它就是 `irq_stack_union` 的基址。

我们还知道， `x86_64` 有一个叫 `中断栈表（Interrupt Stack Table）` 或者 `IST` 的组件，当发生不可屏蔽中断、双重错误等等的时候，这个组件提供了切换到新栈的功能。这可以到达7个 `IST` per-cpu 入口。其中一些如下; There can be up to seven `IST` entries per-cpu. Some of them are:

- `DOUBLEFAULT_STACK`
- `NMI_STACK`
- `DEBUG_STACK`
- `MCE_STACK`

或者

```
#define DOUBLEFAULT_STACK 1
#define NMI_STACK 2
#define DEBUG_STACK 3
#define MCE_STACK 4
```

所有被 `IST` 切换到新栈的中断门描述符都由 `set_intr_gate_ist` 函数初始化。例如:

```
set_intr_gate_ist(X86_TRAP_NMI, &nmi, NMI_STACK);
...
...
...
set_intr_gate_ist(X86_TRAP_DF, &double_fault, DOUBLEFAULT_STACK);
```

其中 `&nmi` 和 `&double_fault` 是中断函数的入口地址：

```
asmlinkage void nmi(void);
asmlinkage void double_fault(void);
```

定义在 arch/x86/kernel/entry_64.S中

```
idtentry double_fault do_double_fault has_error_code=1 paranoid=2
...
...
...
ENTRY(nmi)
...
...
...
END(nmi)
```

当一个中断或者异常发生时，新的 `ss` 选择器被强制置为 `NULL`，并且 `ss` 选择器的 `rpl` 域被设置为新的 `cpl`。旧的 `ss`、`rsp`、寄存器标志、`cs`、`rip` 被压入新栈。在 64 位模型下，中断栈帧大小固定为 8 字节，所以我们可以得到下面的栈：

```
+---------------+
|               |
|      SS       | 40
|      RSP      | 32
|     RFLAGS    | 24
|      CS       | 16
|      RIP      | 8
|   Error code  | 0
|               |
+---------------+
```

如果在中断门中 `IST` 域不是 `0`，我们把 `IST` 读到 `rsp` 中。如果它关联了一个中断向量错误码，我们再把这个错误码压入栈。如果中断向量没有错误码，就继续并且把虚拟错误码压入栈。我们必须做以上的步骤以确保栈一致性。接下来我们从门描述符中加载段选择器域到 CS 寄存器中，并且通过验证第 `21` 位的值来验证目标代码是一个 64 位代码段，例如 `L` 位在 `全局描述符表（Global Descriptor Table）`。最后我们从门描述符中加载偏移域到 `rip` 中，`rip` 是中断处理函数的入口指针。然后中断函数开始执行，在中断函数执行结束后，它必须通过 `iret` 指令把控制权交还给被中断进程。`iret` 指令无条件地弹出栈指针（`ss:rsp`）来恢复被中断的进程，并且不会依赖于 `cpl` 改变。

这就是中断的所有过程。

总结

关于 Linux 内核的中断和中断处理的第一部分至此结束。我们初步了解了一些理论和与中断和异常相关的初始化条件。在下一部分，我会接着深入了解中断和中断处理 - 更深入了解她真实的样子。

如果你有任何问题或建议，请给我发评论或者给我发 Twitter。

请注意英语并不是我的母语，我为任何表达不清楚的地方感到抱歉。如果你发现任何错误请发 **PR** 到 **linux-insides**。**(**译者注：翻译问题请发 **PR** 到 **linux-insides-cn)**

链接

- PIC
- Advanced Programmable Interrupt Controller
- protected mode

- long mode
- kernel stacks
- Task State Segement
- segmented memory model
- Model specific registers
- Stack canary
- Previous chapter

Interrupts and Interrupt Handling. Part 2.

Start to dive into interrupt and exceptions handling in the Linux kernel

We saw some theory about interrupts and exception handling in the previous part and as I already wrote in that part, we will start to dive into interrupts and exceptions in the Linux kernel source code in this part. As you already can note, the previous part mostly described theoretical aspects and in this part we will start to dive directly into the Linux kernel source code. We will start to do it as we did it in other chapters, from the very early places. We will not see the Linux kernel source code from the earliest code lines as we saw it for example in the Linux kernel booting process chapter, but we will start from the earliest code which is related to the interrupts and exceptions. In this part we will try to go through the all interrupts and exceptions related stuff which we can find in the Linux kernel source code.

If you've read the previous parts, you can remember that the earliest place in the Linux kernel `x86_64` architecture-specific source code which is related to the interrupt is located in the arch/x86/boot/pm.c source code file and represents the first setup of the Interrupt Descriptor Table. It occurs right before the transition into the protected mode in the `go_to_protected_mode` function by the call of the `setup_idt` :

```
void go_to_protected_mode(void)
{
    ...
    setup_idt();
    ...
}
```

The `setup_idt` function is defined in the same source code file as the `go_to_protected_mode` function and just loads the address of the `NULL` interrupts descriptor table:

```
static void setup_idt(void)
{
    static const struct gdt_ptr null_idt = {0, 0};
    asm volatile("lidtl %0" : : "m" (null_idt));
}
```

where `gdt_ptr` represents a special 48-bit `GDTR` register which must contain the base address of the `Global Descriptor Table` :

```
struct gdt_ptr {
    u16 len;
    u32 ptr;
} __attribute__((packed));
```

Of course in our case the `gdt_ptr` does not represent the `GDTR` register, but `IDTR` since we set `Interrupt Descriptor Table` . You will not find an `idt_ptr` structure, because if it had been in the Linux kernel source code, it would have been the same as `gdt_ptr` but with different name. So, as you can understand there is no sense to have two similar structures which differ only by name. You can note here, that we do not fill the `Interrupt Descriptor Table` with entries, because it is too early to handle any interrupts or exceptions at this point. That's why we just fill the `IDT` with `NULL` .

After the setup of the Interrupt descriptor table, Global Descriptor Table and other stuff we jump into protected mode in the - arch/x86/boot/pmjump.S. You can read more about it in the part which describes the transition to protected mode.

We already know from the earliest parts that entry to protected mode is located in the `boot_params.hdr.code32_start` and you can see that we pass the entry of the protected mode and `boot_params` to the `protected_mode_jump` in the end of the arch/x86/boot/pm.c:

```
protected_mode_jump(boot_params.hdr.code32_start,
                (u32)&boot_params + (ds() << 4));
```

The `protected_mode_jump` is defined in the arch/x86/boot/pmjump.S and gets these two parameters in the `ax` and `dx` registers using one of the 8086 calling conventions:

```
GLOBAL(protected_mode_jump)
    ...
    ...
    ...
    .byte    0x66, 0xea        # ljmpl opcode
2:    .long    in_pm32            # offset
    .word    __BOOT_CS        # segment
...
...
...
ENDPROC(protected_mode_jump)
```

where `in_pm32` contains a jump to the 32-bit entry point:

```
GLOBAL(in_pm32)
    ...
    ...
    jmpl    *%eax // %eax contains address of the `startup_32`
    ...
    ...
ENDPROC(in_pm32)
```

As you can remember the 32-bit entry point is in the arch/x86/boot/compressed/head_64.S assembly file, although it contains `_64` in its name. We can see the two similar files in the `arch/x86/boot/compressed` directory:

- `arch/x86/boot/compressed/head_32.S`.
- `arch/x86/boot/compressed/head_64.S`;

But the 32-bit mode entry point is the second file in our case. The first file is not even compiled for `x86_64`. Let's look at the arch/x86/boot/compressed/Makefile:

```
vmlinux-objs-y := $(obj)/vmlinux.lds $(obj)/head_$(BITS).o $(obj)/misc.o \
...
...
```

We can see here that `head_*` depends on the `$(BITS)` variable which depends on the architecture. You can find it in the arch/x86/Makefile:

```
ifeq ($(CONFIG_X86_32),y)
...
    BITS := 32
else
    BITS := 64
    ...
endif
```

Now as we jumped on the `startup_32` from the arch/x86/boot/compressed/head_64.S we will not find anything related to the interrupt handling here. The `startup_32` contains code that makes preparations before the transition into long mode and directly jumps in to it. The `long mode` entry is located in `startup_64` and it makes preparations before the kernel decompression that occurs in the `decompress_kernel` from the arch/x86/boot/compressed/misc.c. After the kernel is decompressed, we jump on the `startup_64` from the arch/x86/kernel/head_64.S. In the `startup_64` we start to build identity-mapped pages. After we have built identity-mapped pages, checked the NX bit, setup the `Extended Feature Enable Register` (see in links), and updated the early `Global Descriptor Table` with the `lgdt` instruction, we need to setup `gs` register with the following code:

```
movl    $MSR_GS_BASE,%ecx
movl    initial_gs(%rip),%eax
movl    initial_gs+4(%rip),%edx
wrmsr
```

We already saw this code in the previous part. First of all pay attention on the last `wrmsr` instruction. This instruction writes data from the `edx:eax` registers to the model specific register specified by the `ecx` register. We can see that `ecx` contains `$MSR_GS_BASE` which is declared in the arch/x86/include/uapi/asm/msr-index.h and looks like:

```
#define MSR_GS_BASE             0xc0000101
```

From this we can understand that `MSR_GS_BASE` defines the number of the `model specific register`. Since registers `cs`, `ds`, `es`, and `ss` are not used in the 64-bit mode, their fields are ignored. But we can access memory over `fs` and `gs` registers. The model specific register provides a `back door` to the hidden parts of these segment registers and allows to use 64-bit base address for segment register addressed by the `fs` and `gs`. So the `MSR_GS_BASE` is the hidden part and this part is mapped on the `GS.base` field. Let's look on the `initial_gs`:

```
GLOBAL(initial_gs)
    .quad    INIT_PER_CPU_VAR(irq_stack_union)
```

We pass `irq_stack_union` symbol to the `INIT_PER_CPU_VAR` macro which just concatenates the `init_per_cpu__` prefix with the given symbol. In our case we will get the `init_per_cpu__irq_stack_union` symbol. Let's look at the linker script. There we can see following definition:

```
#define INIT_PER_CPU(x) init_per_cpu__##x = x + __per_cpu_load
INIT_PER_CPU(irq_stack_union);
```

It tells us that the address of the `init_per_cpu__irq_stack_union` will be `irq_stack_union + __per_cpu_load`. Now we need to understand where `init_per_cpu__irq_stack_union` and `__per_cpu_load` are what they mean. The first `irq_stack_union` is defined in the arch/x86/include/asm/processor.h with the `DECLARE_INIT_PER_CPU` macro which expands to call the `init_per_cpu_var` macro:

```
DECLARE_INIT_PER_CPU(irq_stack_union);

#define DECLARE_INIT_PER_CPU(var) \
       extern typeof(per_cpu_var(var)) init_per_cpu_var(var)

#define init_per_cpu_var(var)  init_per_cpu__##var
```

If we expand all macros we will get the same `init_per_cpu__irq_stack_union` as we got after expanding the `INIT_PER_CPU` macro, but you can note that it is not just a symbol, but a variable. Let's look at the `typeof(per_cpu_var(var))` expression. Our `var` is `irq_stack_union` and the `per_cpu_var` macro is defined in the arch/x86/include/asm/percpu.h:

```
#define PER_CPU_VAR(var)        %__percpu_seg:var
```

where:

```
#ifdef CONFIG_X86_64
    #define __percpu_seg gs
endif
```

So, we are accessing `gs:irq_stack_union` and getting its type which is `irq_union`. Ok, we defined the first variable and know its address, now let's look at the second `__per_cpu_load` symbol. There are a couple of `per-cpu` variables which are located after this symbol. The `__per_cpu_load` is defined in the include/asm-generic/sections.h:

```
extern char __per_cpu_load[], __per_cpu_start[], __per_cpu_end[];
```

and presented base address of the `per-cpu` variables from the data area. So, we know the address of the `irq_stack_union`, `__per_cpu_load` and we know that `init_per_cpu__irq_stack_union` must be placed right after `__per_cpu_load`. And we can see it in the System.map:

```
...
...
...
ffffffff819ed000 D __init_begin
ffffffff819ed000 D __per_cpu_load
ffffffff819ed000 A init_per_cpu__irq_stack_union
...
...
...
```

Now we know about `initial_gs` , so let's look at the code:

```
movl    $MSR_GS_BASE,%ecx
movl    initial_gs(%rip),%eax
movl    initial_gs+4(%rip),%edx
wrmsr
```

Here we specified a model specific register with `MSR_GS_BASE` , put the 64-bit address of the `initial_gs` to the `edx:eax` pair and execute the `wrmsr` instruction for filling the `gs` register with the base address of the `init_per_cpu__irq_stack_union` which will be at the bottom of the interrupt stack. After this we will jump to the C code on the `x86_64_start_kernel` from the arch/x86/kernel/head64.c. In the `x86_64_start_kernel` function we do the last preparations before we jump into the generic and architecture-independent kernel code and one of these preparations is filling the early `Interrupt Descriptor Table` with the interrupts handlers entries or `early_idt_handlers` . You can remember it, if you have read the part about the Early interrupt and exception handling and can remember following code:

```
for (i = 0; i < NUM_EXCEPTION_VECTORS; i++)
    set_intr_gate(i, early_idt_handlers[i]);

load_idt((const struct desc_ptr *)&idt_descr);
```

but I wrote `Early interrupt and exception handling` part when Linux kernel version was - `3.18` . For this day actual version of the Linux kernel is `4.1.0-rc6+` and `Andy Lutomirski` sent the patch and soon it will be in the mainline kernel that changes behaviour for the `early_idt_handlers` . **NOTE** While I wrote this part the patch already turned in the Linux kernel source code. Let's look on it. Now the same part looks like:

```
for (i = 0; i < NUM_EXCEPTION_VECTORS; i++)
    set_intr_gate(i, early_idt_handler_array[i]);

load_idt((const struct desc_ptr *)&idt_descr);
```

AS you can see it has only one difference in the name of the array of the interrupts handlers entry points. Now it is `early_idt_handler_arry` :

```
extern const char early_idt_handler_array[NUM_EXCEPTION_VECTORS][EARLY_IDT_HANDLER_SIZ
E];
```

where `NUM_EXCEPTION_VECTORS` and `EARLY_IDT_HANDLER_SIZE` are defined as:

```
#define NUM_EXCEPTION_VECTORS 32
#define EARLY_IDT_HANDLER_SIZE 9
```

So, the `early_idt_handler_array` is an array of the interrupts handlers entry points and contains one entry point on every nine bytes. You can remember that previous `early_idt_handlers` was defined in the arch/x86/kernel/head_64.S. The `early_idt_handler_array` is defined in the same source code file too:

```
ENTRY(early_idt_handler_array)
...
...
...
ENDPROC(early_idt_handler_common)
```

It fills `early_idt_handler_arry` with the `.rept NUM_EXCEPTION_VECTORS` and contains entry of the `early_make_pgtable` interrupt handler (more about its implementation you can read in the part about Early interrupt and exception handling). For now we come to the end of the `x86_64` architecture-specific code and the next part is the generic kernel code. Of course you already can know that we will return to the architecture-specific code in the `setup_arch` function and other places, but this is the end of the `x86_64` early code.

Setting stack canary for the interrupt stack

The next stop after the arch/x86/kernel/head_64.S is the biggest `start_kernel` function from the init/main.c. If you've read the previous chapter about the Linux kernel initialization process, you must remember it. This function does all initialization stuff before kernel will launch first `init` process with the pid - `1`. The first thing that is related to the interrupts and exceptions handling is the call of the `boot_init_stack_canary` function.

This function sets the canary value to protect interrupt stack overflow. We already saw a little some details about implementation of the `boot_init_stack_canary` in the previous part and now let's take a closer look on it. You can find implementation of this function in the arch/x86/include/asm/stackprotector.h and its depends on the `CONFIG_CC_STACKPROTECTOR` kernel configuration option. If this option is not set this function will not do anything:

```
#ifdef CONFIG_CC_STACKPROTECTOR
...
...
...
#else
static inline void boot_init_stack_canary(void)
{
}
#endif
```

If the `CONFIG_CC_STACKPROTECTOR` kernel configuration option is set, the `boot_init_stack_canary` function starts from the check stat `irq_stack_union` that represents per-cpu interrupt stack has offset equal to forty bytes from the `stack_canary` value:

```
#ifdef CONFIG_X86_64
        BUILD_BUG_ON(offsetof(union irq_stack_union, stack_canary) != 40);
#endif
```

As we can read in the previous part the `irq_stack_union` represented by the following union:

```
union irq_stack_union {
    char irq_stack[IRQ_STACK_SIZE];

    struct {
        char gs_base[40];
        unsigned long stack_canary;
    };
};
```

which defined in the arch/x86/include/asm/processor.h. We know that union in the C programming language is a data structure which stores only one field in a memory. We can see here that structure has first field - `gs_base` which is 40 bytes size and represents bottom of the `irq_stack`. So, after this our check with the `BUILD_BUG_ON` macro should end successfully. (you can read the first part about Linux kernel initialization process if you're interesting about the `BUILD_BUG_ON` macro).

After this we calculate new `canary` value based on the random number and Time Stamp Counter:

```
get_random_bytes(&canary, sizeof(canary));
tsc = __native_read_tsc();
canary += tsc + (tsc << 32UL);
```

and write `canary` value to the `irq_stack_union` with the `this_cpu_write` macro:

```
this_cpu_write(irq_stack_union.stack_canary, canary);
```

more about `this_cpu_*` operation you can read in the Linux kernel documentation.

Disabling/Enabling local interrupts

The next step in the init/main.c which is related to the interrupts and interrupts handling after we have set the `canary` value to the interrupt stack - is the call of the `local_irq_disable` macro.

This macro defined in the include/linux/irqflags.h header file and as you can understand, we can disable interrupts for the CPU with the call of this macro. Let's look on its implementation. First of all note that it depends on the `CONFIG_TRACE_IRQFLAGS_SUPPORT` kernel configuration option:

```
#ifdef CONFIG_TRACE_IRQFLAGS_SUPPORT
...
#define local_irq_disable() \
         do { raw_local_irq_disable(); trace_hardirqs_off(); } while (0)
...
#else
...
#define local_irq_disable()     do { raw_local_irq_disable(); } while (0)
...
#endif
```

They are both similar and as you can see have only one difference: the `local_irq_disable` macro contains call of the `trace_hardirqs_off` when `CONFIG_TRACE_IRQFLAGS_SUPPORT` is enabled. There is special feature in the lockdep subsystem - `irq-flags tracing` for tracing `hardirq` and `softirq` state. In our case `lockdep` subsystem can give us interesting information about hard/soft irqs on/off events which are occurs in the system. The `trace_hardirqs_off` function defined in the kernel/locking/lockdep.c:

```
void trace_hardirqs_off(void)
{
         trace_hardirqs_off_caller(CALLER_ADDR0);
}
EXPORT_SYMBOL(trace_hardirqs_off);
```

and just calls `trace_hardirqs_off_caller` function. The `trace_hardirqs_off_caller` checks the `hardirqs_enabled` field of the current process and increases the `redundant_hardirqs_off` if call of the `local_irq_disable` was redundant or the `hardirqs_off_events` if it was not. These two fields and other `lockdep` statistic related fields are defined in the kernel/locking/lockdep_insides.h and located in the `lockdep_stats` structure:

```
struct lockdep_stats {
...
...
...
int     softirqs_off_events;
int     redundant_softirqs_off;
...
...
...
}
```

If you will set `CONFIG_DEBUG_LOCKDEP` kernel configuration option, the `lockdep_stats_debug_show` function will write all tracing information to the `/proc/lockdep` :

```
static void lockdep_stats_debug_show(struct seq_file *m)
{
#ifdef CONFIG_DEBUG_LOCKDEP
    unsigned long long hi1 = debug_atomic_read(hardirqs_on_events),
                         hi2 = debug_atomic_read(hardirqs_off_events),
                         hr1 = debug_atomic_read(redundant_hardirqs_on),
    ...
    ...
    ...
    seq_printf(m, " hardirq on events:             %11llu\n", hi1);
    seq_printf(m, " hardirq off events:            %11llu\n", hi2);
    seq_printf(m, " redundant hardirq ons:         %11llu\n", hr1);
#endif
}
```

and you can see its result with the:

```
$ sudo cat /proc/lockdep
 hardirq on events:             12838248974
 hardirq off events:            12838248979
 redundant hardirq ons:               67792
 redundant hardirq offs:         3836339146
 softirq on events:                38002159
 softirq off events:               38002187
 redundant softirq ons:                   0
 redundant softirq offs:                  0
```

Ok, now we know a little about tracing, but more info will be in the separate part about `lockdep` and `tracing`. You can see that the both `local_disable_irq` macros have the same part - `raw_local_irq_disable`. This macro defined in the arch/x86/include/asm/irqflags.h and expands to the call of the:

```
static inline void native_irq_disable(void)
{
        asm volatile("cli": : :"memory");
}
```

And you already must remember that `cli` instruction clears the IF flag which determines ability of a processor to handle an interrupt or an exception. Besides the `local_irq_disable`, as you already can know there is an inverse macro - `local_irq_enable`. This macro has the same tracing mechanism and very similar on the `local_irq_enable`, but as you can understand from its name, it enables interrupts with the `sti` instruction:

```
static inline void native_irq_enable(void)
{
        asm volatile("sti": : :"memory");
}
```

Now we know how `local_irq_disable` and `local_irq_enable` work. It was the first call of the `local_irq_disable` macro, but we will meet these macros many times in the Linux kernel source code. But for now we are in the `start_kernel` function from the init/main.c and we just disabled `local` interrupts. Why local and why we did it? Previously kernel provided a method to disable interrupts on all processors and it was called `cli`. This function was removed and now we have `local_irq_{enabled,disable}` to disable or enable interrupts on the current processor. After we've disabled the interrupts with the `local_irq_disable` macro, we set the:

```
early_boot_irqs_disabled = true;
```

The `early_boot_irqs_disabled` variable defined in the include/linux/kernel.h:

```
extern bool early_boot_irqs_disabled;
```

and used in the different places. For example it used in the `smp_call_function_many` function from the kernel/smp.c for the checking possible deadlock when interrupts are disabled:

```
WARN_ON_ONCE(cpu_online(this_cpu) && irqs_disabled()
                     && !oops_in_progress && !early_boot_irqs_disabled);
```

Early trap initialization during kernel initialization

The next functions after the `local_disable_irq` are `boot_cpu_init` and `page_address_init`, but they are not related to the interrupts and exceptions (more about this functions you can read in the chapter about Linux kernel initialization process). The next is the `setup_arch` function. As you can remember this function located in the arch/x86/kernel/setup.c source code file and makes initialization of many different architecture-dependent stuff. The first interrupts related function which we can see in the `setup_arch` is the - `early_trap_init` function. This function defined in the arch/x86/kernel/traps.c and fills `Interrupt Descriptor Table` with the couple of entries:

```
void __init early_trap_init(void)
{
        set_intr_gate_ist(X86_TRAP_DB, &debug, DEBUG_STACK);
        set_system_intr_gate_ist(X86_TRAP_BP, &int3, DEBUG_STACK);
#ifdef CONFIG_X86_32
        set_intr_gate(X86_TRAP_PF, page_fault);
#endif
        load_idt(&idt_descr);
}
```

Here we can see calls of three different functions:

- `set_intr_gate_ist`
- `set_system_intr_gate_ist`
- `set_intr_gate`

All of these functions defined in the arch/x86/include/asm/desc.h and do the similar thing but not the same. The first `set_intr_gate_ist` function inserts new an interrupt gate in the `IDT`. Let's look on its implementation:

```
static inline void set_intr_gate_ist(int n, void *addr, unsigned ist)
{
        BUG_ON((unsigned)n > 0xFF);
        _set_gate(n, GATE_INTERRUPT, addr, 0, ist, __KERNEL_CS);
}
```

First of all we can see the check that `n` which is vector number of the interrupt is not greater than `0xff` or 255. We need to check it because we remember from the previous part that vector number of an interrupt must be between `0` and `255`. In the next step we can see the call of the `_set_gate` function that sets a given interrupt gate to the `IDT` table:

```
static inline void _set_gate(int gate, unsigned type, void *addr,
                             unsigned dpl, unsigned ist, unsigned seg)
{
        gate_desc s;

        pack_gate(&s, type, (unsigned long)addr, dpl, ist, seg);
        write_idt_entry(idt_table, gate, &s);
        write_trace_idt_entry(gate, &s);
}
```

Here we start from the `pack_gate` function which takes clean `IDT` entry represented by the `gate_desc` structure and fills it with the base address and limit, Interrupt Stack Table, Privilege level, type of an interrupt which can be one of the following values:

- `GATE_INTERRUPT`
- `GATE_TRAP`
- `GATE_CALL`
- `GATE_TASK`

and set the present bit for the given `IDT` entry:

```
static inline void pack_gate(gate_desc *gate, unsigned type, unsigned long func,
                             unsigned dpl, unsigned ist, unsigned seg)
{
        gate->offset_low        = PTR_LOW(func);
        gate->segment           = __KERNEL_CS;
        gate->ist               = ist;
        gate->p                 = 1;
        gate->dpl               = dpl;
        gate->zero0             = 0;
        gate->zero1             = 0;
        gate->type              = type;
        gate->offset_middle     = PTR_MIDDLE(func);
        gate->offset_high       = PTR_HIGH(func);
}
```

After this we write just filled interrupt gate to the `IDT` with the `write_idt_entry` macro which expands to the `native_write_idt_entry` and just copy the interrupt gate to the `idt_table` table by the given index:

```
#define write_idt_entry(dt, entry, g)           native_write_idt_entry(dt, entry, g)

static inline void native_write_idt_entry(gate_desc *idt, int entry, const gate_desc *
gate)
{
        memcpy(&idt[entry], gate, sizeof(*gate));
}
```

where `idt_table` is just array of `gate_desc` :

```
extern gate_desc idt_table[];
```

That's all. The second `set_system_intr_gate_ist` function has only one difference from the `set_intr_gate_ist` :

```
static inline void set_system_intr_gate_ist(int n, void *addr, unsigned ist)
{
        BUG_ON((unsigned)n > 0xFF);
        _set_gate(n, GATE_INTERRUPT, addr, 0x3, ist, __KERNEL_CS);
}
```

Do you see it? Look on the fourth parameter of the `_set_gate` . It is `0x3` . In the `set_intr_gate` it was `0x0` . We know that this parameter represent `DPL` or privilege level. We also know that `0` is the highest privilege level and `3` is the lowest.Now we know how `set_system_intr_gate_ist` , `set_intr_gate_ist` , `set_intr_gate` are work and we can return to the `early_trap_init` function. Let's look on it again:

```
set_intr_gate_ist(X86_TRAP_DB, &debug, DEBUG_STACK);
set_system_intr_gate_ist(X86_TRAP_BP, &int3, DEBUG_STACK);
```

We set two `IDT` entries for the `#DB` interrupt and `int3` . These functions takes the same set of parameters:

- vector number of an interrupt;
- address of an interrupt handler;
- interrupt stack table index.

That's all. More about interrupts and handlers you will know in the next parts.

Conclusion

It is the end of the second part about interrupts and interrupt handling in the Linux kernel. We saw the some theory in the previous part and started to dive into interrupts and exceptions handling in the current part. We have started from the earliest parts in the Linux kernel source code which are related to the interrupts. In the next part we will continue to dive into this interesting theme and will know more about interrupt handling process.

If you have any questions or suggestions write me a comment or ping me at twitter.

Please note that English is not my first language, And I am really sorry for any inconvenience. If you find any mistakes please send me PR to linux-insides.

Links

- IDT
- Protected mode
- List of x86 calling conventions
- 8086
- Long mode
- NX
- Extended Feature Enable Register
- Model-specific register
- Process identifier
- lockdep
- irqflags tracing
- IF
- Stack canary
- Union type
- this*cpu** operations
- vector number
- Interrupt Stack Table
- Privilege level
- Previous part

Interrupts and Interrupt Handling. Part 3.

Exception Handling

This is the third part of the chapter about an interrupts and an exceptions handling in the Linux kernel and in the previous part we stopped at the `setup_arch` function from the arch/x86/kernel/setup.c source code file.

We already know that this function executes initialization of architecture-specific stuff. In our case the `setup_arch` function does x86_64 architecture related initializations. The `setup_arch` is big function, and in the previous part we stopped on the setting of the two exceptions handlers for the two following exceptions:

- `#DB` - debug exception, transfers control from the interrupted process to the debug handler;
- `#BP` - breakpoint exception, caused by the `int 3` instruction.

These exceptions allow the `x86_64` architecture to have early exception processing for the purpose of debugging via the kgdb.

As you can remember we set these exceptions handlers in the `early_trap_init` function:

```
void __init early_trap_init(void)
{
        set_intr_gate_ist(X86_TRAP_DB, &debug, DEBUG_STACK);
        set_system_intr_gate_ist(X86_TRAP_BP, &int3, DEBUG_STACK);
        load_idt(&idt_descr);
}
```

from the arch/x86/kernel/traps.c. We already saw implementation of the `set_intr_gate_ist` and `set_system_intr_gate_ist` functions in the previous part and now we will look on the implementation of these two exceptions handlers.

Debug and Breakpoint exceptions

Ok, we setup exception handlers in the `early_trap_init` function for the `#DB` and `#BP` exceptions and now time is to consider their implementations. But before we will do this, first of all let's look on details of these exceptions.

The first exceptions - `#DB` or `debug` exception occurs when a debug event occurs. For example - attempt to change the contents of a debug register. Debug registers are special registers that were presented in `x86` processors starting from the Intel 80386 processor and as you can understand from name of this CPU extension, main purpose of these registers is debugging.

These registers allow to set breakpoints on the code and read or write data to trace it. Debug registers may be accessed only in the privileged mode and an attempt to read or write the debug registers when executing at any other privilege level causes a general protection fault exception. That's why we have used `set_intr_gate_ist` for the `#DB` exception, but not the `set_system_intr_gate_ist` .

The verctor number of the `#DB` exceptions is `1` (we pass it as `X86_TRAP_DB`) and as we may read in specification, this exception has no error code:

```
+-----------------------------------------------------+
|Vector|Mnemonic|Description         |Type |Error Code|
+-----------------------------------------------------+
|1     | #DB    |Reserved            |F/T  |NO        |
+-----------------------------------------------------+
```

The second exception is `#BP` or `breakpoint` exception occurs when processor executes the int 3 instruction. Unlike the `DB` exception, the `#BP` exception may occur in userspace. We can add it anywhere in our code, for example let's look on the simple program:

```
// breakpoint.c
#include <stdio.h>

int main() {
    int i;
    while (i < 6){
        printf("i equal to: %d\n", i);
        __asm__("int3");
        ++i;
    }
}
```

If we will compile and run this program, we will see following output:

```
$ gcc breakpoint.c -o breakpoint
i equal to: 0
Trace/breakpoint trap
```

But if will run it with gdb, we will see our breakpoint and can continue execution of our program:

```
$ gdb breakpoint
...
...
...
(gdb) run
Starting program: /home/alex/breakpoints
i equal to: 0

Program received signal SIGTRAP, Trace/breakpoint trap.
0x0000000000400585 in main ()
=> 0x0000000000400585 <main+31>:    83 45 fc 01    add    DWORD PTR [rbp-0x4],0x1
(gdb) c
Continuing.
i equal to: 1

Program received signal SIGTRAP, Trace/breakpoint trap.
0x0000000000400585 in main ()
=> 0x0000000000400585 <main+31>:    83 45 fc 01    add    DWORD PTR [rbp-0x4],0x1
(gdb) c
Continuing.
i equal to: 2

Program received signal SIGTRAP, Trace/breakpoint trap.
0x0000000000400585 in main ()
=> 0x0000000000400585 <main+31>:    83 45 fc 01    add    DWORD PTR [rbp-0x4],0x1
...
...
...
```

From this moment we know a little about these two exceptions and we can move on to consideration of their handlers.

Preparation before an exception handler

As you may note before, the `set_intr_gate_ist` and `set_system_intr_gate_ist` functions takes an addresses of exceptions handlers in theirs second parameter. In or case our two exception handlers will be:

* `debug` ;
* `int3` .

You will not find these functions in the C code. all of that could be found in the kernel's `*.c/*.h` files only definition of these functions which are located in the arch/x86/include/asm/traps.h kernel header file:

```
asmlinkage void debug(void);
```

and

```
asmlinkage void int3(void);
```

You may note `asmlinkage` directive in definitions of these functions. The directive is the special specificator of the gcc. Actually for a `C` functions which are called from assembly, we need in explicit declaration of the function calling convention. In our case, if function made with `asmlinkage` descriptor, then `gcc` will compile the function to retrieve parameters from stack.

So, both handlers are defined in the arch/x86/entry/entry_64.S assembly source code file with the `idtentry` macro:

```
idtentry debug do_debug has_error_code=0 paranoid=1 shift_ist=DEBUG_STACK
```

and

```
idtentry int3 do_int3 has_error_code=0 paranoid=1 shift_ist=DEBUG_STACK
```

Each exception handler may be consists from two parts. The first part is generic part and it is the same for all exception handlers. An exception handler should to save general purpose registers on the stack, switch to kernel stack if an exception came from userspace and transfer control to the second part of an exception handler. The second part of an exception handler does certain work depends on certain exception. For example page fault exception handler should find virtual page for given address, invalid opcode exception handler should send `SIGILL` signal and etc.

As we just saw, an exception handler starts from definition of the `idtentry` macro from the arch/x86/kernel/entry_64.S assembly source code file, so let's look at implementation of this macro. As we may see, the `idtentry` macro takes five arguments:

- `sym` - defines global symbol with the `.globl name` which will be an an entry of exception handler;
- `do_sym` - symbol name which represents a secondary entry of an exception handler;
- `has_error_code` - information about existence of an error code of exception.

The last two parameters are optional:

- `paranoid` - shows us how we need to check current mode (will see explanation in details later);
- `shift_ist` - shows us is an exception running at `Interrupt Stack Table`.

Definition of the `.idtentry` macro looks:

```
.macro idtentry sym do_sym has_error_code:req paranoid=0 shift_ist=-1
ENTRY(\sym)
...
...
...
END(\sym)
.endm
```

Before we will consider internals of the `idtentry` macro, we should to know state of stack when an exception occurs. As we may read in the Intel® 64 and IA-32 Architectures Software Developer's Manual 3A, the state of stack when an exception occurs is following:

```
    +------------+
+40 | %SS        |
+32 | %RSP       |
+24 | %RFLAGS    |
+16 | %CS        |
 +8 | %RIP       |
  0 | ERROR CODE | <-- %RSP
    +------------+
```

Now we may start to consider implementation of the `idtmacro`. Both `#DB` and `BP` exception handlers are defined as:

```
idtentry debug do_debug has_error_code=0 paranoid=1 shift_ist=DEBUG_STACK
idtentry int3 do_int3 has_error_code=0 paranoid=1 shift_ist=DEBUG_STACK
```

If we will look at these definitions, we may know that compiler will generate two routines with `debug` and `int3` names and both of these exception handlers will call `do_debug` and `do_int3` secondary handlers after some preparation. The third parameter defines existence of error code and as we may see both our exception do not have them. As we may see on the diagram above, processor pushes error code on stack if an exception provides it. In our case, the `debug` and `int3` exception do not have error codes. This may bring some difficulties because stack will look differently for exceptions which provides error code and for exceptions which not. That's why implementation of the `idtentry` macro starts from putting a fake error code to the stack if an exception does not provide it:

```
.ifeq \has_error_code
    pushq    $-1
.endif
```

But it is not only fake error-code. Moreover the `-1` also represents invalid system call number, so that the system call restart logic will not be triggered.

The last two parameters of the `idtentry` macro `shift_ist` and `paranoid` allow to know do an exception handler runned at stack from `Interrupt Stack Table` or not. You already may know that each kernel thread in the system has own stack. In addition to these stacks, there are some specialized stacks associated with each processor in the system. One of these stacks is - exception stack. The x86_64 architecture provides special feature which is called - `Interrupt Stack Table`. This feature allows to switch to a new stack for designated events such as an atomic exceptions like `double fault` and etc. So the `shift_ist` parameter allows us to know do we need to switch on `IST` stack for an exception handler or not.

The second parameter - `paranoid` defines the method which helps us to know did we come from userspace or not to an exception handler. The easiest way to determine this is to via `CPL` or `Current Privilege Level` in `CS` segment register. If it is equal to `3`, we came from userspace, if zero we came from kernel space:

```
testl $3,CS(%rsp)
jnz userspace
...
...
...
// we are from the kernel space
```

But unfortunately this method does not give a 100% guarantee. As described in the kernel documentation:

> if we are in an NMI/MCE/DEBUG/whatever super-atomic entry context, which might have triggered right after a normal entry wrote CS to the stack but before we executed SWAPGS, then the only safe way to check for GS is the slower method: the RDMSR.

In other words for example `NMI` could happen inside the critical section of a swapgs instruction. In this way we should check value of the `MSR_GS_BASE` model specific register which stores pointer to the start of per-cpu area. So to check did we come from userspace or not, we should to check value of the `MSR_GS_BASE` model specific register and if it is negative we came from kernel space, in other way we came from userspace:

```
movl $MSR_GS_BASE,%ecx
rdmsr
testl %edx,%edx
js 1f
```

In first two lines of code we read value of the `MSR_GS_BASE` model specific register into `edx:eax` pair. We can't set negative value to the `gs` from userspace. But from other side we know that direct mapping of the physical memory starts from the `0xffff880000000000` virtual address. In this way, `MSR_GS_BASE` will contain an address from `0xffff880000000000`

to `0xffffc7ffffffffff` . After the `rdmsr` instruction will be executed, the smallest possible value in the `%edx` register will be - `0xffff8800` which is `-30720` in unsigned 4 bytes. That's why kernel space `gs` which points to start of `per-cpu` area will contain negative value.

After we pushed fake error code on the stack, we should allocate space for general purpose registers with:

```
ALLOC_PT_GPREGS_ON_STACK
```

macro which is defined in the arch/x86/entry/calling.h header file. This macro just allocates 15*8 bytes space on the stack to preserve general purpose registers:

```
.macro ALLOC_PT_GPREGS_ON_STACK addskip=0
    addq    $-(15*8+\addskip), %rsp
.endm
```

So the stack will look like this after execution of the `ALLOC_PT_GPREGS_ON_STACK` :

```
     +------------+
+160 | %SS        |
+152 | %RSP       |
+144 | %RFLAGS    |
+136 | %CS        |
+128 | %RIP       |
+120 | ERROR CODE |
     |------------|
+112 |            |
+104 |            |
 +96 |            |
 +88 |            |
 +80 |            |
 +72 |            |
 +64 |            |
 +56 |            |
 +48 |            |
 +40 |            |
 +32 |            |
 +24 |            |
 +16 |            |
  +8 |            |
  +0 |            | <- %RSP
     +------------+
```

After we allocated space for general purpose registers, we do some checks to understand did an exception come from userspace or not and if yes, we should move back to an interrupted process stack or stay on exception stack:

```
.if \paranoid
    .if \paranoid == 1
        testb   $3, CS(%rsp)
        jnz     1f
    .endif
    call    paranoid_entry
.else
    call    error_entry
.endif
```

Let's consider all of these there cases in course.

An exception occured in userspace

In the first let's consider a case when an exception has `paranoid=1` like our `debug` and `int3` exceptions. In this case we check selector from `CS` segment register and jump at `1f` label if we came from userspace or the `paranoid_entry` will be called in other way.

Let's consider first case when we came from userspace to an exception handler. As described above we should jump at `1` label. The `1` label starts from the call of the

```
call    error_entry
```

routine which saves all general purpose registers in the previously allocated area on the stack:

```
SAVE_C_REGS 8
SAVE_EXTRA_REGS 8
```

These both macros are defined in the arch/x86/entry/calling.h header file and just move values of general purpose registers to a certain place at the stack, for example:

```
.macro SAVE_EXTRA_REGS offset=0
    movq %r15, 0*8+\offset(%rsp)
    movq %r14, 1*8+\offset(%rsp)
    movq %r13, 2*8+\offset(%rsp)
    movq %r12, 3*8+\offset(%rsp)
    movq %rbp, 4*8+\offset(%rsp)
    movq %rbx, 5*8+\offset(%rsp)
.endm
```

After execution of `SAVE_C_REGS` and `SAVE_EXTRA_REGS` the stack will look:

```
      +------------+
+160 | %SS         |
+152 | %RSP        |
+144 | %RFLAGS     |
+136 | %CS         |
+128 | %RIP        |
+120 | ERROR CODE  |
      |------------|
+112 | %RDI        |
+104 | %RSI        |
 +96 | %RDX        |
 +88 | %RCX        |
 +80 | %RAX        |
 +72 | %R8         |
 +64 | %R9         |
 +56 | %R10        |
 +48 | %R11        |
 +40 | %RBX        |
 +32 | %RBP        |
 +24 | %R12        |
 +16 | %R13        |
  +8 | %R14        |
  +0 | %R15        | <- %RSP
      +------------+
```

After the kernel saved general purpose registers at the stack, we should check that we came from userspace space again with:

```
testb    $3, CS+8(%rsp)
jz    .Lerror_kernelspace
```

because we may have potentially fault if as described in documentation truncated `%RIP` was reported. Anyway, in both cases the SWAPGS instruction will be executed and values from `MSR_KERNEL_GS_BASE` and `MSR_GS_BASE` will be swapped. From this moment the `%gs` register will point to the base address of kernel structures. So, the `SWAPGS` instruction is called and it was main point of the `error_entry` routing.

Now we can back to the `idtentry` macro. We may see following assembler code after the call of `error_entry` :

```
movq    %rsp, %rdi
call    sync_regs
```

Here we put base address of stack pointer `%rdi` register which will be first argument (according to x86_64 ABI) of the `sync_regs` function and call this function which is defined in the arch/x86/kernel/traps.c source code file:

```
asmlinkage __visible notrace struct pt_regs *sync_regs(struct pt_regs *eregs)
{
    struct pt_regs *regs = task_pt_regs(current);
    *regs = *eregs;
    return regs;
}
```

This function takes the result of the `task_ptr_regs` macro which is defined in the arch/x86/include/asm/processor.h header file, stores it in the stack pointer and return it. The `task_ptr_regs` macro expands to the address of `thread.sp0` which represents pointer to the normal kernel stack:

```
#define task_pt_regs(tsk)       ((struct pt_regs *)(tsk)->thread.sp0 - 1)
```

As we came from userspace, this means that exception handler will run in real process context. After we got stack pointer from the `sync_regs` we switch stack:

```
movq    %rax, %rsp
```

The last two steps before an exception handler will call secondary handler are:

1. Passing pointer to `pt_regs` structure which contains preserved general purpose registers to the `%rdi` register:

```
movq    %rsp, %rdi
```

as it will be passed as first parameter of secondary exception handler.

1. Pass error code to the `%rsi` register as it will be second argument of an exception handler and set it to `-1` on the stack for the same purpose as we did it before - to prevent restart of a system call:

```
.if \has_error_code
    movq    ORIG_RAX(%rsp), %rsi
    movq    $-1, ORIG_RAX(%rsp)
.else
    xorl    %esi, %esi
.endif
```

Additionally you may see that we zeroed the `%esi` register above in a case if an exception does not provide error code.

In the end we just call secondary exception handler:

```
call    \do_sym
```

which:

```
dotraplinkage void do_debug(struct pt_regs *regs, long error_code);
```

will be for `debug` exception and:

```
dotraplinkage void notrace do_int3(struct pt_regs *regs, long error_code);
```

will be for `int 3` exception. In this part we will not see implementations of secondary handlers, because of they are very specific, but will see some of them in one of next parts.

We just considered first case when an exception occurred in userspace. Let's consider last two.

An exception with paranoid > 0 occurred in kernelspace

In this case an exception was occurred in kernelspace and `idtentry` macro is defined with `paranoid=1` for this exception. This value of `paranoid` means that we should use slower way that we saw in the beginning of this part to check do we really came from kernelspace or not. The `paranoid_entry` routing allows us to know this:

```
ENTRY(paranoid_entry)
    cld
    SAVE_C_REGS 8
    SAVE_EXTRA_REGS 8
    movl    $1, %ebx
    movl    $MSR_GS_BASE, %ecx
    rdmsr
    testl    %edx, %edx
    js    1f
    SWAPGS
    xorl    %ebx, %ebx
1:    ret
END(paranoid_entry)
```

As you may see, this function represents the same that we covered before. We use second (slow) method to get information about previous state of an interrupted task. As we checked this and executed `SWAPGS` in a case if we came from userspace, we should to do the same

that we did before: We need to put pointer to a structure which holds general purpose registers to the `%rdi` (which will be first parameter of a secondary handler) and put error code if an exception provides it to the `%rsi` (which will be second parameter of a secondary handler):

```
movq    %rsp, %rdi

.if \has_error_code
    movq    ORIG_RAX(%rsp), %rsi
    movq    $-1, ORIG_RAX(%rsp)
.else
    xorl    %esi, %esi
.endif
```

The last step before a secondary handler of an exception will be called is cleanup of new `IST` stack fram:

```
.if \shift_ist != -1
    subq    $EXCEPTION_STKSZ, CPU_TSS_IST(\shift_ist)
.endif
```

You may remember that we passed the `shift_ist` as argument of the `idtentry` macro. Here we check its value and if its not equal to `-1`, we get pointer to a stack from `Interrupt Stack Table` by `shift_ist` index and setup it.

In the end of this second way we just call secondary exception handler as we did it before:

```
call    \do_sym
```

The last method is similar to previous both, but an exception occured with `paranoid=0` and we may use fast method determination of where we are from.

Exit from an exception handler

After secondary handler will finish its works, we will return to the `idtentry` macro and the next step will be jump to the `error_exit`:

```
jmp    error_exit
```

routine. The `error_exit` function defined in the same arch/x86/entry/entry_64.S assembly source code file and the main goal of this function is to know where we are from (from userspace or kernelspace) and execute `SWPAGS` depends on this. Restore registers to

previous state and execute `iret` instruction to transfer control to an interrupted task.

That's all.

Conclusion

It is the end of the third part about interrupts and interrupt handling in the Linux kernel. We saw the initialization of the Interrupt descriptor table in the previous part with the `#DB` and `#BP` gates and started to dive into preparation before control will be transferred to an exception handler and implementation of some interrupt handlers in this part. In the next part we will continue to dive into this theme and will go next by the `setup_arch` function and will try to understand interrupts handling related stuff.

If you have any questions or suggestions write me a comment or ping me at twitter.

Please note that English is not my first language, And I am really sorry for any inconvenience. If you find any mistakes please send me PR to linux-insides.

Links

- Debug registers
- Intel 80385
- INT 3
- gcc
- TSS
- GNU assembly .error directive
- dwarf2
- CFI directives
- IRQ
- system call
- swapgs
- SIGTRAP
- Per-CPU variables
- kgdb
- ACPI
- Previous part

Interrupts and Interrupt Handling. Part 4.

Initialization of non-early interrupt gates

This is fourth part about an interrupts and exceptions handling in the Linux kernel and in the previous part we saw first early `#DB` and `#BP` exceptions handlers from the arch/x86/kernel/traps.c. We stopped on the right after the `early_trap_init` function that called in the `setup_arch` function which defined in the arch/x86/kernel/setup.c. In this part we will continue to dive into an interrupts and exceptions handling in the Linux kernel for `x86_64` and continue to do it from the place where we left off in the last part. First thing which is related to the interrupts and exceptions handling is the setup of the `#PF` or page fault handler with the `early_trap_pf_init` function. Let's start from it.

Early page fault handler

The `early_trap_pf_init` function defined in the arch/x86/kernel/traps.c. It uses `set_intr_gate` macro that fills Interrupt Descriptor Table with the given entry:

```
void __init early_trap_pf_init(void)
{
#ifdef CONFIG_X86_64
         set_intr_gate(X86_TRAP_PF, page_fault);
#endif
}
```

This macro defined in the arch/x86/include/asm/desc.h. We already saw macros like this in the previous part - `set_system_intr_gate` and `set_intr_gate_ist`. This macro checks that given vector number is not greater than `255` (maximum vector number) and calls `_set_gate` function as `set_system_intr_gate` and `set_intr_gate_ist` did it:

```
#define set_intr_gate(n, addr)                                  \
do {                                                            \
        BUG_ON((unsigned)n > 0xFF);                             \
        _set_gate(n, GATE_INTERRUPT, (void *)addr, 0, 0,        \
                  __KERNEL_CS);                                 \
        _trace_set_gate(n, GATE_INTERRUPT, (void *)trace_##addr,\
                        0, 0, __KERNEL_CS);                     \
} while (0)
```

The `set_intr_gate` macro takes two parameters:

- vector number of a interrupt;
- address of an interrupt handler;

In our case they are:

- `X86_TRAP_PF` - `14` ;
- `page_fault` - the interrupt handler entry point.

The `X86_TRAP_PF` is the element of enum which defined in the arch/x86/include/asm/traprs.h:

```
enum {
    ...
    ...
    ...
    ...
    X86_TRAP_PF,            /* 14, Page Fault */
    ...
    ...
    ...
}
```

When the `early_trap_pf_init` will be called, the `set_intr_gate` will be expanded to the call of the `_set_gate` which will fill the `IDT` with the handler for the page fault. Now let's look on the implementation of the `page_fault` handler. The `page_fault` handler defined in the arch/x86/kernel/entry_64.S assembly source code file as all exceptions handlers. Let's look on it:

```
trace_idtentry page_fault do_page_fault has_error_code=1
```

We saw in the previous part how `#DB` and `#BP` handlers defined. They were defined with the `idtentry` macro, but here we can see `trace_idtentry`. This macro defined in the same source code file and depends on the `CONFIG_TRACING` kernel configuration option:

```
#ifdef CONFIG_TRACING
.macro trace_idtentry sym do_sym has_error_code:req
idtentry trace(\sym) trace(\do_sym) has_error_code=\has_error_code
idtentry \sym \do_sym has_error_code=\has_error_code
.endm
#else
.macro trace_idtentry sym do_sym has_error_code:req
idtentry \sym \do_sym has_error_code=\has_error_code
.endm
#endif
```

We will not dive into exceptions Tracing now. If `CONFIG_TRACING` is not set, we can see that `trace_idtentry` macro just expands to the normal `idtentry`. We already saw implementation of the `idtentry` macro in the previous part, so let's start from the `page_fault` exception handler.

As we can see in the `idtentry` definition, the handler of the `page_fault` is `do_page_fault` function which defined in the arch/x86/mm/fault.c and as all exceptions handlers it takes two arguments:

- `regs` - `pt_regs` structure that holds state of an interrupted process;
- `error_code` - error code of the page fault exception.

Let's look inside this function. First of all we read content of the cr2 control register:

```
dotraplinkage void notrace
do_page_fault(struct pt_regs *regs, unsigned long error_code)
{
    unsigned long address = read_cr2();
    ...
    ...
    ...
}
```

This register contains a linear address which caused `page fault`. In the next step we make a call of the `exception_enter` function from the include/linux/context_tracking.h. The `exception_enter` and `exception_exit` are functions from context tracking subsystem in the Linux kernel used by the RCU to remove its dependency on the timer tick while a processor runs in userspace. Almost in the every exception handler we will see similar code:

```
enum ctx_state prev_state;
prev_state = exception_enter();
...
... // exception handler here
...
exception_exit(prev_state);
```

The `exception_enter` function checks that `context tracking` is enabled with the `context_tracking_is_enabled` and if it is in enabled state, we get previous context with the `this_cpu_read` (more about `this_cpu_*` operations you can read in the Documentation). After this it calls `context_tracking_user_exit` function which informs the context tracking that the processor is exiting userspace mode and entering the kernel:

```
static inline enum ctx_state exception_enter(void)
{
        enum ctx_state prev_ctx;

        if (!context_tracking_is_enabled())
                return 0;

        prev_ctx = this_cpu_read(context_tracking.state);
        context_tracking_user_exit();

        return prev_ctx;
}
```

The state can be one of the:

```
enum ctx_state {
    IN_KERNEL = 0,
    IN_USER,
} state;
```

And in the end we return previous context. Between the `exception_enter` and `exception_exit` we call actual page fault handler:

```
__do_page_fault(regs, error_code, address);
```

The `__do_page_fault` is defined in the same source code file as `do_page_fault` - arch/x86/mm/fault.c. In the beginning of the `__do_page_fault` we check state of the kmemcheck checker. The `kmemcheck` detects warns about some uses of uninitialized memory. We need to check it because page fault can be caused by kmemcheck:

```
if (kmemcheck_active(regs))
        kmemcheck_hide(regs);
    prefetchw(&mm->mmap_sem);
```

After this we can see the call of the `prefetchw` which executes instruction with the same name which fetches X86_FEATURE_3DNOW to get exclusive cache line. The main purpose of prefetching is to hide the latency of a memory access. In the next step we check that we got page fault not in the kernel space with the following condition:

```
if (unlikely(fault_in_kernel_space(address))) {
...
...
...
}
```

where `fault_in_kernel_space` is:

```
static int fault_in_kernel_space(unsigned long address)
{
        return address >= TASK_SIZE_MAX;
}
```

The `TASK_SIZE_MAX` macro expands to the:

```
#define TASK_SIZE_MAX   ((1UL << 47) - PAGE_SIZE)
```

or `0x00007ffffffff000`. Pay attention on `unlikely` macro. There are two macros in the Linux kernel:

```
#define likely(x)      __builtin_expect(!!(x), 1)
#define unlikely(x)    __builtin_expect(!!(x), 0)
```

You can often find these macros in the code of the Linux kernel. Main purpose of these macros is optimization. Sometimes this situation is that we need to check the condition of the code and we know that it will rarely be `true` or `false`. With these macros we can tell to the compiler about this. For example

```
static int proc_root_readdir(struct file *file, struct dir_context *ctx)
{
        if (ctx->pos < FIRST_PROCESS_ENTRY) {
                int error = proc_readdir(file, ctx);
                if (unlikely(error <= 0))
                        return error;
...
...
...
}
```

Here we can see `proc_root_readdir` function which will be called when the Linux VFS needs to read the `root` directory contents. If condition marked with `unlikely`, compiler can put `false` code right after branching. Now let's back to the our address check. Comparison between the given address and the `0x00007ffffffff000` will give us to know, was page fault in the kernel mode or user mode. After this check we know it. After this `__do_page_fault` routine will try to understand the problem that provoked page fault exception and then will pass address to the appropriate routine. It can be `kmemcheck` fault, spurious fault, kprobes fault and etc. Will not dive into implementation details of the page

fault exception handler in this part, because we need to know many different concepts which are provided by the Linux kernel, but will see it in the chapter about the memory management in the Linux kernel.

Back to start_kernel

There are many different function calls after the `early_trap_pf_init` in the `setup_arch` function from different kernel subsystems, but there are no one interrupts and exceptions handling related. So, we have to go back where we came from - `start_kernel` function from the init/main.c. The first things after the `setup_arch` is the `trap_init` function from the arch/x86/kernel/traps.c. This function makes initialization of the remaining exceptions handlers (remember that we already setup 3 handlers for the `#DB` - debug exception, `#BP` - breakpoint exception and `#PF` - page fault exception). The `trap_init` function starts from the check of the Extended Industry Standard Architecture:

```
#ifdef CONFIG_EISA
        void __iomem *p = early_ioremap(0x0FFFD9, 4);

        if (readl(p) == 'E' + ('I'<<8) + ('S'<<16) + ('A'<<24))
                EISA_bus = 1;
        early_iounmap(p, 4);
#endif
```

Note that it depends on the `CONFIG_EISA` kernel configuration parameter which represents `EISA` support. Here we use `early_ioremap` function to map `I/O` memory on the page tables. We use `readl` function to read first `4` bytes from the mapped region and if they are equal to `EISA` string we set `EISA_bus` to one. In the end we just unmap previously mapped region. More about `early_ioremap` you can read in the part which describes Fix-Mapped Addresses and ioremap.

After this we start to fill the `Interrupt Descriptor Table` with the different interrupt gates. First of all we set `#DE` or `Divide Error` and `#NMI` or `Non-maskable Interrupt`:

```
set_intr_gate(X86_TRAP_DE, divide_error);
set_intr_gate_ist(X86_TRAP_NMI, &nmi, NMI_STACK);
```

We use `set_intr_gate` macro to set the interrupt gate for the `#DE` exception and `set_intr_gate_ist` for the `#NMI`. You can remember that we already used these macros when we have set the interrupts gates for the page fault handler, debug handler and etc, you can find explanation of it in the previous part. After this we setup exception gates for the following exceptions:

```
set_system_intr_gate(X86_TRAP_OF, &overflow);
set_intr_gate(X86_TRAP_BR, bounds);
set_intr_gate(X86_TRAP_UD, invalid_op);
set_intr_gate(X86_TRAP_NM, device_not_available);
```

Here we can see:

- `#OF` or `Overflow` exception. This exception indicates that an overflow trap occurred when an special INTO instruction was executed;
- `#BR` or `BOUND Range exceeded` exception. This exception indicates that a `BOUND-range-exceed` fault occurred when a BOUND instruction was executed;
- `#UD` or `Invalid Opcode` exception. Occurs when a processor attempted to execute invalid or reserved opcode, processor attempted to execute instruction with invalid operand(s) and etc;
- `#NM` or `Device Not Available` exception. Occurs when the processor tries to execute `x87 FPU` floating point instruction while `EM` flag in the control register `cr0` was set.

In the next step we set the interrupt gate for the `#DF` or `Double fault` exception:

```
set_intr_gate_ist(X86_TRAP_DF, &double_fault, DOUBLEFAULT_STACK);
```

This exception occurs when processor detected a second exception while calling an exception handler for a prior exception. In usual way when the processor detects another exception while trying to call an exception handler, the two exceptions can be handled serially. If the processor cannot handle them serially, it signals the double-fault or `#DF` exception.

The following set of the interrupt gates is:

```
set_intr_gate(X86_TRAP_OLD_MF, &coprocessor_segment_overrun);
set_intr_gate(X86_TRAP_TS, &invalid_TSS);
set_intr_gate(X86_TRAP_NP, &segment_not_present);
set_intr_gate_ist(X86_TRAP_SS, &stack_segment, STACKFAULT_STACK);
set_intr_gate(X86_TRAP_GP, &general_protection);
set_intr_gate(X86_TRAP_SPURIOUS, &spurious_interrupt_bug);
set_intr_gate(X86_TRAP_MF, &coprocessor_error);
set_intr_gate(X86_TRAP_AC, &alignment_check);
```

Here we can see setup for the following exception handlers:

- `#CSO` or `Coprocessor Segment Overrun` - this exception indicates that math coprocessor of an old processor detected a page or segment violation. Modern processors do not generate this exception
- `#TS` or `Invalid TSS` exception - indicates that there was an error related to the Task

State Segment.
- `#NP` or `Segment Not Present` exception indicates that the `present flag` of a segment or gate descriptor is clear during attempt to load one of `cs`, `ds`, `es`, `fs`, or `gs` register.
- `#SS` or `Stack Fault` exception indicates one of the stack related conditions was detected, for example a not-present stack segment is detected when attempting to load the `ss` register.
- `#GP` or `General Protection` exception indicates that the processor detected one of a class of protection violations called general-protection violations. There are many different conditions that can cause general-protection exception. For example loading the `ss`, `ds`, `es`, `fs`, or `gs` register with a segment selector for a system segment, writing to a code segment or a read-only data segment, referencing an entry in the `Interrupt Descriptor Table` (following an interrupt or exception) that is not an interrupt, trap, or task gate and many many more.
- `Spurious Interrupt` - a hardware interrupt that is unwanted.
- `#MF` or `x87 FPU Floating-Point Error` exception caused when the x87 FPU has detected a floating point error.
- `#AC` or `Alignment Check` exception Indicates that the processor detected an unaligned memory operand when alignment checking was enabled.

After that we setup this exception gates, we can see setup of the `Machine-Check` exception:

```
#ifdef CONFIG_X86_MCE
    set_intr_gate_ist(X86_TRAP_MC, &machine_check, MCE_STACK);
#endif
```

Note that it depends on the `CONFIG_X86_MCE` kernel configuration option and indicates that the processor detected an internal machine error or a bus error, or that an external agent detected a bus error. The next exception gate is for the SIMD Floating-Point exception:

```
set_intr_gate(X86_TRAP_XF, &simd_coprocessor_error);
```

which indicates the processor has detected an `SSE` or `SSE2` or `SSE3` SIMD floating-point exception. There are six classes of numeric exception conditions that can occur while executing an SIMD floating-point instruction:

- Invalid operation
- Divide-by-zero
- Denormal operand
- Numeric overflow
- Numeric underflow

- Inexact result (Precision)

In the next step we fill the `used_vectors` array which defined in the arch/x86/include/asm/desc.h header file and represents `bitmap` :

```
DECLARE_BITMAP(used_vectors, NR_VECTORS);
```

of the first `32` interrupts (more about bitmaps in the Linux kernel you can read in the part which describes cpumasks and bitmaps)

```
for (i = 0; i < FIRST_EXTERNAL_VECTOR; i++)
    set_bit(i, used_vectors)
```

where `FIRST_EXTERNAL_VECTOR` is:

```
#define FIRST_EXTERNAL_VECTOR           0x20
```

After this we setup the interrupt gate for the `ia32_syscall` and add `0x80` to the `used_vectors` bitmap:

```
#ifdef CONFIG_IA32_EMULATION
        set_system_intr_gate(IA32_SYSCALL_VECTOR, ia32_syscall);
        set_bit(IA32_SYSCALL_VECTOR, used_vectors);
#endif
```

There is `CONFIG_IA32_EMULATION` kernel configuration option on `x86_64` Linux kernels. This option provides ability to execute 32-bit processes in compatibility-mode. In the next parts we will see how it works, in the meantime we need only to know that there is yet another interrupt gate in the `IDT` with the vector number `0x80` . In the next step we maps `IDT` to the fixmap area:

```
__set_fixmap(FIX_RO_IDT, __pa_symbol(idt_table), PAGE_KERNEL_RO);
idt_descr.address = fix_to_virt(FIX_RO_IDT);
```

and write its address to the `idt_descr.address` (more about fix-mapped addresses you can read in the second part of the Linux kernel memory management chapter). After this we can see the call of the `cpu_init` function that defined in the arch/x86/kernel/cpu/common.c. This function makes initialization of the all `per-cpu` state. In the beginning of the `cpu_init` we do the following things: First of all we wait while current cpu is initialized and than we call the `cr4_init_shadow` function which stores shadow copy of the `cr4` control register for the current cpu and load CPU microcode if need with the following function calls:

```
wait_for_master_cpu(cpu);
cr4_init_shadow();
load_ucode_ap();
```

Next we get the `Task State Segment` for the current cpu and `orig_ist` structure which represents origin `Interrupt Stack Table` values with the:

```
t = &per_cpu(cpu_tss, cpu);
oist = &per_cpu(orig_ist, cpu);
```

As we got values of the `Task State Segment` and `Interrupt Stack Table` for the current processor, we clear following bits in the `cr4` control register:

```
cr4_clear_bits(X86_CR4_VME|X86_CR4_PVI|X86_CR4_TSD|X86_CR4_DE);
```

with this we disable `vm86` extension, virtual interrupts, timestamp (RDTSC can only be executed with the highest privilege) and debug extension. After this we reload the `Global Descriptor Table` and `Interrupt Descriptor table` with the:

```
    switch_to_new_gdt(cpu);
    loadsegment(fs, 0);
    load_current_idt();
```

After this we setup array of the Thread-Local Storage Descriptors, configure NX and load CPU microcode. Now is time to setup and load `per-cpu` Task State Segments. We are going in a loop through the all exception stack which is `N_EXCEPTION_STACKS` or `4` and fill it with `Interrupt Stack Tables`:

```
    if (!oist->ist[0]) {
        char *estacks = per_cpu(exception_stacks, cpu);

        for (v = 0; v < N_EXCEPTION_STACKS; v++) {
            estacks += exception_stack_sizes[v];
            oist->ist[v] = t->x86_tss.ist[v] =
                    (unsigned long)estacks;
            if (v == DEBUG_STACK-1)
                per_cpu(debug_stack_addr, cpu) = (unsigned long)estacks;
        }
    }
```

As we have filled `Task State Segments` with the `Interrupt Stack Tables` we can set `TSS` descriptor for the current processor and load it with the:

```
set_tss_desc(cpu, t);
load_TR_desc();
```

where `set_tss_desc` macro from the arch/x86/include/asm/desc.h writes given descriptor to the `Global Descriptor Table` of the given processor:

```
#define set_tss_desc(cpu, addr) __set_tss_desc(cpu, GDT_ENTRY_TSS, addr)
static inline void __set_tss_desc(unsigned cpu, unsigned int entry, void *addr)
{
        struct desc_struct *d = get_cpu_gdt_table(cpu);
        tss_desc tss;
        set_tssldt_descriptor(&tss, (unsigned long)addr, DESC_TSS,
                              IO_BITMAP_OFFSET + IO_BITMAP_BYTES +
                              sizeof(unsigned long) - 1);
        write_gdt_entry(d, entry, &tss, DESC_TSS);
}
```

and `load_TR_desc` macro expands to the `ltr` or `Load Task Register` instruction:

```
#define load_TR_desc()                          native_load_tr_desc()
static inline void native_load_tr_desc(void)
{
        asm volatile("ltr %w0"::"q" (GDT_ENTRY_TSS*8));
}
```

In the end of the `trap_init` function we can see the following code:

```
set_intr_gate_ist(X86_TRAP_DB, &debug, DEBUG_STACK);
set_system_intr_gate_ist(X86_TRAP_BP, &int3, DEBUG_STACK);
...
...
...
#ifdef CONFIG_X86_64
        memcpy(&nmi_idt_table, &idt_table, IDT_ENTRIES * 16);
        set_nmi_gate(X86_TRAP_DB, &debug);
        set_nmi_gate(X86_TRAP_BP, &int3);
#endif
```

Here we copy `idt_table` to the `nmi_dit_table` and setup exception handlers for the `#DB` or `Debug exception` and `#BR` or `Breakpoint exception`. You can remember that we already set these interrupt gates in the previous part, so why do we need to setup it again? We setup it again because when we initialized it before in the `early_trap_init` function, the `Task State Segment` was not ready yet, but now it is ready after the call of the `cpu_init` function.

That's all. Soon we will consider all handlers of these interrupts/exceptions.

Conclusion

It is the end of the fourth part about interrupts and interrupt handling in the Linux kernel. We saw the initialization of the Task State Segment in this part and initialization of the different interrupt handlers as `Divide Error` , `Page Fault` exception and etc. You can note that we saw just initialization stuff, and will dive into details about handlers for these exceptions. In the next part we will start to do it.

If you have any questions or suggestions write me a comment or ping me at twitter.

Please note that English is not my first language, And I am really sorry for any inconvenience. If you find any mistakes please send me PR to linux-insides.

Links

- page fault
- Interrupt Descriptor Table
- Tracing
- cr2
- RCU
- this*cpu** operations
- kmemcheck
- prefetchw
- 3DNow
- CPU caches
- VFS
- Linux kernel memory management
- Fix-Mapped Addresses and ioremap
- Extended Industry Standard Architecture
- INT isntruction
- INTO
- BOUND
- opcode
- control register
- x87 FPU
- MCE exception
- SIMD
- cpumasks and bitmaps

- NX
- Task State Segment
- Previous part

Interrupts and Interrupt Handling. Part 5.

Implementation of exception handlers

This is the fifth part about an interrupts and exceptions handling in the Linux kernel and in the previous part we stopped on the setting of interrupt gates to the Interrupt descriptor Table. We did it in the `trap_init` function from the arch/x86/kernel/traps.c source code file. We saw only setting of these interrupt gates in the previous part and in the current part we will see implementation of the exception handlers for these gates. The preparation before an exception handler will be executed is in the arch/x86/entry/entry_64.S assembly file and occurs in the idtentry macro that defines exceptions entry points:

```
idtentry divide_error                    do_divide_error                  has_error_c
ode=0
idtentry overflow                        do_overflow                      has_error_c
ode=0
idtentry invalid_op                        do_invalid_op                  has_error_c
ode=0
idtentry bounds                            do_bounds                      has_error_c
ode=0
idtentry device_not_available            do_device_not_available          has_error_c
ode=0
idtentry coprocessor_segment_overrun    do_coprocessor_segment_overrun has_error_code=
0
idtentry invalid_TSS                    do_invalid_TSS                  has_error_cod
e=1
idtentry segment_not_present            do_segment_not_present          has_error_cod
e=1
idtentry spurious_interrupt_bug            do_spurious_interrupt_bug      has_error_c
ode=0
idtentry coprocessor_error               do_coprocessor_error           has_error_cod
e=0
idtentry alignment_check                do_alignment_check              has_error_cod
e=1
idtentry simd_coprocessor_error            do_simd_coprocessor_error      has_error_c
ode=0
```

The `idtentry` macro does following preparation before an actual exception handler (`do_divide_error` for the `divide_error` , `do_overflow` for the `overflow` and etc.) will get control. In another words the `idtentry` macro allocates place for the registers (pt_regs structure) on the stack, pushes dummy error code for the stack consistency if an

interrupt/exception has no error code, checks the segment selector in the `cs` segment register and switches depends on the previous state(userspace or kernelspace). After all of these preparations it makes a call of an actual interrupt/exception handler:

```
.macro idtentry sym do_sym has_error_code:req paranoid=0 shift_ist=-1
ENTRY(\sym)
	...
	...
	...
	call	\do_sym
	...
	...
	...
END(\sym)
.endm
```

After an exception handler will finish its work, the `idtentry` macro restores stack and general purpose registers of an interrupted task and executes iret instruction:

```
ENTRY(paranoid_exit)
	...
	...
	...
	RESTORE_EXTRA_REGS
	RESTORE_C_REGS
	REMOVE_PT_GPREGS_FROM_STACK 8
	INTERRUPT_RETURN
END(paranoid_exit)
```

where `INTERRUPT_RETURN` is:

```
#define INTERRUPT_RETURN	jmp native_iret
...
ENTRY(native_iret)
.global native_irq_return_iret
native_irq_return_iret:
iretq
```

More about the `idtentry` macro you can read in the third part of the http://0xax.gitbooks.io/linux-insides/content/interrupts/interrupts-3.html chapter. Ok, now we saw the preparation before an exception handler will be executed and now time to look on the handlers. First of all let's look on the following handlers:

* divide_error
* overflow
* invalid_op

- coprocessor_segment_overrun
- invalid_TSS
- segment_not_present
- stack_segment
- alignment_check

All these handlers defined in the arch/x86/kernel/traps.c source code file with the `DO_ERROR` macro:

```
DO_ERROR(X86_TRAP_DE,     SIGFPE,  "divide error",                divide_error)
DO_ERROR(X86_TRAP_OF,     SIGSEGV, "overflow",                    overflow)
DO_ERROR(X86_TRAP_UD,     SIGILL,  "invalid opcode",              invalid_op)
DO_ERROR(X86_TRAP_OLD_MF, SIGFPE,  "coprocessor segment overrun", coprocessor_segment_
overrun)
DO_ERROR(X86_TRAP_TS,     SIGSEGV, "invalid TSS",                 invalid_TSS)
DO_ERROR(X86_TRAP_NP,     SIGBUS,  "segment not present",         segment_not_present)
DO_ERROR(X86_TRAP_SS,     SIGBUS,  "stack segment",               stack_segment)
DO_ERROR(X86_TRAP_AC,     SIGBUS,  "alignment check",             alignment_check)
```

As we can see the `DO_ERROR` macro takes 4 parameters:

- Vector number of an interrupt;
- Signal number which will be sent to the interrupted process;
- String which describes an exception;
- Exception handler entry point.

This macro defined in the same source code file and expands to the function with the `do_handler` name:

```
#define DO_ERROR(trapnr, signr, str, name)                              \
dotraplinkage void do_##name(struct pt_regs *regs, long error_code)     \
{                                                                       \
        do_error_trap(regs, error_code, str, trapnr, signr);            \
}
```

Note on the `##` tokens. This is special feature - GCC macro Concatenation which concatenates two given strings. For example, first `DO_ERROR` in our example will expands to the:

```
dotraplinkage void do_divide_error(struct pt_regs *regs, long error_code)     \
{
    ...
}
```

We can see that all functions which are generated by the `DO_ERROR` macro just make a call of the `do_error_trap` function from the arch/x86/kernel/traps.c. Let's look on implementation of the `do_error_trap` function.

Trap handlers

The `do_error_trap` function starts and ends from the two following functions:

```
enum ctx_state prev_state = exception_enter();
...
...
...
exception_exit(prev_state);
```

from the include/linux/context_tracking.h. The context tracking in the Linux kernel subsystem which provide kernel boundaries probes to keep track of the transitions between level contexts with two basic initial contexts: `user` or `kernel`. The `exception_enter` function checks that context tracking is enabled. After this if it is enabled, the `exception_enter` reads previous context and compares it with the `CONTEXT_KERNEL`. If the previous context is `user`, we call `context_tracking_exit` function from the kernel/context_tracking.c which inform the context tracking subsystem that a processor is exiting user mode and entering the kernel mode:

```
if (!context_tracking_is_enabled())
    return 0;

prev_ctx = this_cpu_read(context_tracking.state);
if (prev_ctx != CONTEXT_KERNEL)
    context_tracking_exit(prev_ctx);

return prev_ctx;
```

If previous context is non `user`, we just return it. The `pre_ctx` has `enum ctx_state` type which defined in the include/linux/context_tracking_state.h and looks as:

```
enum ctx_state {
    CONTEXT_KERNEL = 0,
    CONTEXT_USER,
    CONTEXT_GUEST,
} state;
```

The second function is `exception_exit` defined in the same include/linux/context_tracking.h file and checks that context tracking is enabled and call the `contert_tracking_enter` function if the previous context was `user` :

```
static inline void exception_exit(enum ctx_state prev_ctx)
{
    if (context_tracking_is_enabled()) {
        if (prev_ctx != CONTEXT_KERNEL)
            context_tracking_enter(prev_ctx);
    }
}
```

The `context_tracking_enter` function informs the context tracking subsystem that a processor is going to enter to the user mode from the kernel mode. We can see the following code between the `exception_enter` and `exception_exit` :

```
if (notify_die(DIE_TRAP, str, regs, error_code, trapnr, signr) !=
        NOTIFY_STOP) {
    conditional_sti(regs);
    do_trap(trapnr, signr, str, regs, error_code,
        fill_trap_info(regs, signr, trapnr, &info));
}
```

First of all it calls the `notify_die` function which defined in the kernel/notifier.c. To get notified for kernel panic, kernel oops, Non-Maskable Interrupt or other events the caller needs to insert itself in the `notify_die` chain and the `notify_die` function does it. The Linux kernel has special mechanism that allows kernel to ask when something happens and this mechanism called `notifiers` or `notifier chains` . This mechanism used for example for the `USB` hotplug events (look on the drivers/usb/core/notify.c), for the memory hotplug (look on the include/linux/memory.h, the `hotplug_memory_notifier` macro and etc...), system reboots and etc. A notifier chain is thus a simple, singly-linked list. When a Linux kernel subsystem wants to be notified of specific events, it fills out a special `notifier_block` structure and passes it to the `notifier_chain_register` function. An event can be sent with the call of the `notifier_call_chain` function. First of all the `notify_die` function fills `die_args` structure with the trap number, trap string, registers and other values:

```
struct die_args args = {
       .regs   = regs,
       .str    = str,
       .err    = err,
       .trapnr = trap,
       .signr  = sig,
}
```

and returns the result of the `atomic_notifier_call_chain` function with the `die_chain` :

```
static ATOMIC_NOTIFIER_HEAD(die_chain);
return atomic_notifier_call_chain(&die_chain, val, &args);
```

which just expands to the `atomic_notifier_head` structure that contains lock and `notifier_block` :

```
struct atomic_notifier_head {
        spinlock_t lock;
        struct notifier_block __rcu *head;
};
```

The `atomic_notifier_call_chain` function calls each function in a notifier chain in turn and returns the value of the last notifier function called. If the `notify_die` in the `do_error_trap` does not return `NOTIFY_STOP` we execute `conditional_sti` function from the arch/x86/kernel/traps.c that checks the value of the interrupt flag and enables interrupt depends on it:

```
static inline void conditional_sti(struct pt_regs *regs)
{
        if (regs->flags & X86_EFLAGS_IF)
                local_irq_enable();
}
```

more about `local_irq_enable` macro you can read in the second part of this chapter. The next and last call in the `do_error_trap` is the `do_trap` function. First of all the `do_trap` function defined the `tsk` variable which has `task_struct` type and represents the current interrupted process. After the definition of the `tsk` , we can see the call of the `do_trap_no_signal` function:

```
struct task_struct *tsk = current;

if (!do_trap_no_signal(tsk, trapnr, str, regs, error_code))
    return;
```

The `do_trap_no_signal` function makes two checks:

- Did we come from the Virtual 8086 mode;
- Did we come from the kernelspace.

```
if (v8086_mode(regs)) {
    ...
}

if (!user_mode(regs)) {
    ...
}

return -1;
```

We will not consider first case because the long mode does not support the Virtual 8086 mode. In the second case we invoke `fixup_exception` function which will try to recover a fault and `die` if we can't:

```
if (!fixup_exception(regs)) {
    tsk->thread.error_code = error_code;
    tsk->thread.trap_nr = trapnr;
    die(str, regs, error_code);
}
```

The `die` function defined in the arch/x86/kernel/dumpstack.c source code file, prints useful information about stack, registers, kernel modules and caused kernel oops. If we came from the userspace the `do_trap_no_signal` function will return `-1` and the execution of the `do_trap` function will continue. If we passed through the `do_trap_no_signal` function and did not exit from the `do_trap` after this, it means that previous context was - `user`. Most exceptions caused by the processor are interpreted by Linux as error conditions, for example division by zero, invalid opcode and etc. When an exception occurs the Linux kernel sends a signal to the interrupted process that caused the exception to notify it of an incorrect condition. So, in the `do_trap` function we need to send a signal with the given number (`SIGFPE` for the divide error, `SIGILL` for the overflow exception and etc...). First of all we save error code and vector number in the current interrupts process with the filling `thread.error_code` and `thread_trap_nr` :

```
tsk->thread.error_code = error_code;
tsk->thread.trap_nr = trapnr;
```

After this we make a check do we need to print information about unhandled signals for the interrupted process. We check that `show_unhandled_signals` variable is set, that `unhandled_signal` function from the kernel/signal.c will return unhandled signal(s) and printk rate limit:

```
#ifdef CONFIG_X86_64
    if (show_unhandled_signals && unhandled_signal(tsk, signr) &&
        printk_ratelimit()) {
        pr_info("%s[%d] trap %s ip:%lx sp:%lx error:%lx",
            tsk->comm, tsk->pid, str,
            regs->ip, regs->sp, error_code);
        print_vma_addr(" in ", regs->ip);
        pr_cont("\n");
    }
#endif
```

And send a given signal to interrupted process:

```
force_sig_info(signr, info ?: SEND_SIG_PRIV, tsk);
```

This is the end of the `do_trap`. We just saw generic implementation for eight different exceptions which are defined with the `DO_ERROR` macro. Now let's look on another exception handlers.

Double fault

The next exception is `#DF` or `Double fault`. This exception occurs when the processor detected a second exception while calling an exception handler for a prior exception. We set the trap gate for this exception in the previous part:

```
set_intr_gate_ist(X86_TRAP_DF, &double_fault, DOUBLEFAULT_STACK);
```

Note that this exception runs on the `DOUBLEFAULT_STACK` Interrupt Stack Table which has index - `1`:

```
#define DOUBLEFAULT_STACK 1
```

The `double_fault` is handler for this exception and defined in the arch/x86/kernel/traps.c. The `double_fault` handler starts from the definition of two variables: string that describes exception and interrupted process, as other exception handlers:

```
static const char str[] = "double fault";
struct task_struct *tsk = current;
```

The handler of the double fault exception split on two parts. The first part is the check which checks that a fault is a `non-IST` fault on the `espfix64` stack. Actually the `iret` instruction restores only the bottom `16` bits when returning to a `16` bit segment. The `espfix` feature solves this problem. So if the `non-IST` fault on the espfix64 stack we modify the stack to make it look like `General Protection Fault` :

```
struct pt_regs *normal_regs = task_pt_regs(current);

memmove(&normal_regs->ip, (void *)regs->sp, 5*8);
ormal_regs->orig_ax = 0;
regs->ip = (unsigned long)general_protection;
regs->sp = (unsigned long)&normal_regs->orig_ax;
return;
```

In the second case we do almost the same that we did in the previous exception handlers. The first is the call of the `ist_enter` function that discards previous context, `user` in our case:

```
ist_enter(regs);
```

And after this we fill the interrupted process with the vector number of the `Double fault` exception and error code as we did it in the previous handlers:

```
tsk->thread.error_code = error_code;
tsk->thread.trap_nr = X86_TRAP_DF;
```

Next we print useful information about the double fault (PID number, registers content):

```
#ifdef CONFIG_DOUBLEFAULT
    df_debug(regs, error_code);
#endif
```

And die:

```
    for (;;)
        die(str, regs, error_code);
```

That's all.

Device not available exception handler

The next exception is the `#NM` or `Device not available`. The `Device not available` exception can occur depending on these things:

- The processor executed an x87 FPU floating-point instruction while the EM flag in control register `cr0` was set;
- The processor executed a `wait` or `fwait` instruction while the `MP` and `TS` flags of register `cr0` were set;
- The processor executed an x87 FPU, MMX or SSE instruction while the `TS` flag in control register `cr0` was set and the `EM` flag is clear.

The handler of the `Device not available` exception is the `do_device_not_available` function and it defined in the arch/x86/kernel/traps.c source code file too. It starts and ends from the getting of the previous context, as other traps which we saw in the beginning of this part:

```
enum ctx_state prev_state;
prev_state = exception_enter();
...
...
...
exception_exit(prev_state);
```

In the next step we check that `FPU` is not eager:

```
BUG_ON(use_eager_fpu());
```

When we switch into a task or interrupt we may avoid loading the `FPU` state. If a task will use it, we catch `Device not Available exception` exception. If we loading the `FPU` state during task switching, the `FPU` is eager. In the next step we check `cr0` control register on the `EM` flag which can show us is `x87` floating point unit present (flag clear) or not (flag set):

```
#ifdef CONFIG_MATH_EMULATION
    if (read_cr0() & X86_CR0_EM) {
        struct math_emu_info info = { };

        conditional_sti(regs);

        info.regs = regs;
        math_emulate(&info);
        exception_exit(prev_state);
        return;
    }
#endif
```

If the `x87` floating point unit not presented, we enable interrupts with the `conditional_sti`, fill the `math_emu_info` (defined in the arch/x86/include/asm/math_emu.h) structure with the registers of an interrupt task and call `math_emulate` function from the arch/x86/math-emu/fpu_entry.c. As you can understand from function's name, it emulates `X87 FPU` unit (more about the `x87` we will know in the special chapter). In other way, if `X86_CR0_EM` flag is clear which means that `x87 FPU` unit is presented, we call the `fpu__restore` function from the arch/x86/kernel/fpu/core.c which copies the `FPU` registers from the `fpustate` to the live hardware registers. After this `FPU` instructions can be used:

```
fpu__restore(&current->thread.fpu);
```

General protection fault exception handler

The next exception is the `#GP` or `General protection fault`. This exception occurs when the processor detected one of a class of protection violations called `general-protection violations`. It can be:

- Exceeding the segment limit when accessing the `cs`, `ds`, `es`, `fs` or `gs` segments;
- Loading the `ss`, `ds`, `es`, `fs` or `gs` register with a segment selector for a system segment.;
- Violating any of the privilege rules;
- and other...

The exception handler for this exception is the `do_general_protection` from the arch/x86/kernel/traps.c. The `do_general_protection` function starts and ends as other exception handlers from the getting of the previous context:

```
prev_state = exception_enter();
...
exception_exit(prev_state);
```

After this we enable interrupts if they were disabled and check that we came from the Virtual 8086 mode:

```
conditional_sti(regs);

if (v8086_mode(regs)) {
    local_irq_enable();
    handle_vm86_fault((struct kernel_vm86_regs *) regs, error_code);
    goto exit;
}
```

As long mode does not support this mode, we will not consider exception handling for this case. In the next step check that previous mode was kernel mode and try to fix the trap. If we can't fix the current general protection fault exception we fill the interrupted process with the vector number and error code of the exception and add it to the `notify_die` chain:

```
if (!user_mode(regs)) {
    if (fixup_exception(regs))
        goto exit;

    tsk->thread.error_code = error_code;
    tsk->thread.trap_nr = X86_TRAP_GP;
    if (notify_die(DIE_GPF, "general protection fault", regs, error_code,
               X86_TRAP_GP, SIGSEGV) != NOTIFY_STOP)
        die("general protection fault", regs, error_code);
    goto exit;
}
```

If we can fix exception we go to the `exit` label which exits from exception state:

```
exit:
    exception_exit(prev_state);
```

If we came from user mode we send `SIGSEGV` signal to the interrupted process from user mode as we did it in the `do_trap` function:

```
if (show_unhandled_signals && unhandled_signal(tsk, SIGSEGV) &&
        printk_ratelimit()) {
    pr_info("%s[%d] general protection ip:%lx sp:%lx error:%lx",
        tsk->comm, task_pid_nr(tsk),
        regs->ip, regs->sp, error_code);
    print_vma_addr(" in ", regs->ip);
    pr_cont("\n");
}

force_sig_info(SIGSEGV, SEND_SIG_PRIV, tsk);
```

That's all.

Conclusion

It is the end of the fifth part of the Interrupts and Interrupt Handling chapter and we saw implementation of some interrupt handlers in this part. In the next part we will continue to dive into interrupt and exception handlers and will see handler for the Non-Maskable

Interrupts, handling of the math coprocessor and SIMD coprocessor exceptions and many many more.

If you have any questions or suggestions write me a comment or ping me at twitter.

Please note that English is not my first language, And I am really sorry for any inconvenience. If you find any mistakes please send me PR to linux-insides.

Links

- Interrupt descriptor Table
- iret instruction
- GCC macro Concatenation
- kernel panic
- kernel oops
- Non-Maskable Interrupt
- hotplug
- interrupt flag
- long mode
- signal
- printk
- coprocessor
- SIMD
- Interrupt Stack Table
- PID
- x87 FPU
- control register
- MMX
- Previous part

Interrupts and Interrupt Handling. Part 6.

Non-maskable interrupt handler

It is sixth part of the Interrupts and Interrupt Handling in the Linux kernel chapter and in the previous part we saw implementation of some exception handlers for the General Protection Fault exception, divide exception, invalid opcode exceptions and etc. As I wrote in the previous part we will see implementations of the rest exceptions in this part. We will see implementation of the following handlers:

- Non-Maskable interrupt;
- BOUND Range Exceeded Exception;
- Coprocessor exception;
- SIMD coprocessor exception.

in this part. So, let's start.

Non-Maskable interrupt handling

A Non-Maskable interrupt is a hardware interrupt that cannot be ignored by standard masking techniques. In a general way, a non-maskable interrupt can be generated in either of two ways:

- External hardware asserts the non-maskable interrupt pin on the CPU.
- The processor receives a message on the system bus or the APIC serial bus with a delivery mode `NMI` .

When the processor receives a `NMI` from one of these sources, the processor handles it immediately by calling the `NMI` handler pointed to by interrupt vector which has number `2` (see table in the first part). We already filled the Interrupt Descriptor Table with the vector number, address of the `nmi` interrupt handler and `NMI_STACK` Interrupt Stack Table entry:

```
set_intr_gate_ist(X86_TRAP_NMI, &nmi, NMI_STACK);
```

in the `trap_init` function which defined in the arch/x86/kernel/traps.c source code file. In the previous parts we saw that entry points of the all interrupt handlers are defined with the:

```
.macro idtentry sym do_sym has_error_code:req paranoid=0 shift_ist=-1
ENTRY(\sym)
...
...
...
END(\sym)
.endm
```

macro from the arch/x86/entry/entry_64.S assembly source code file. But the handler of the `Non-Maskable` interrupts is not defined with this macro. It has own entry point:

```
ENTRY(nmi)
...
...
...
END(nmi)
```

in the same arch/x86/entry/entry_64.S assembly file. Lets dive into it and will try to understand how `Non-Maskable` interrupt handler works. The `nmi` handlers starts from the call of the:

```
PARAVIRT_ADJUST_EXCEPTION_FRAME
```

macro but we will not dive into details about it in this part, because this macro related to the Paravirtualization stuff which we will see in another chapter. After this save the content of the `rdx` register on the stack:

```
pushq    %rdx
```

And allocated check that `cs` was not the kernel segment when an non-maskable interrupt occurs:

```
cmpl    $__KERNEL_CS, 16(%rsp)
jne    first_nmi
```

The `__KERNEL_CS` macro defined in the arch/x86/include/asm/segment.h and represented second descriptor in the Global Descriptor Table:

```
#define GDT_ENTRY_KERNEL_CS    2
#define __KERNEL_CS    (GDT_ENTRY_KERNEL_CS*8)
```

more about `GDT` you can read in the second part of the Linux kernel booting process chapter. If `cs` is not kernel segment, it means that it is not nested `NMI` and we jump on the `first_nmi` label. Let's consider this case. First of all we put address of the current stack pointer to the `rdx` and pushes `1` to the stack in the `first_nmi` label:

```
first_nmi:
    movq    (%rsp), %rdx
    pushq    $1
```

Why do we push `1` on the stack? As the comment says: `We allow breakpoints in NMIs`. On the x86_64, like other architectures, the CPU will not execute another `NMI` until the first `NMI` is completed. A `NMI` interrupt finished with the iret instruction like other interrupts and exceptions do it. If the `NMI` handler triggers either a page fault or breakpoint or another exception which are use `iret` instruction too. If this happens while in `NMI` context, the CPU will leave `NMI` context and a new `NMI` may come in. The `iret` used to return from those exceptions will re-enable `NMIs` and we will get nested non-maskable interrupts. The problem the `NMI` handler will not return to the state that it was, when the exception triggered, but instead it will return to a state that will allow new `NMIs` to preempt the running `NMI` handler. If another `NMI` comes in before the first NMI handler is complete, the new NMI will write all over the preempted `NMIs` stack. We can have nested `NMIs` where the next `NMI` is using the top of the stack of the previous `NMI`. It means that we cannot execute it because a nested non-maskable interrupt will corrupt stack of a previous non-maskable interrupt. That's why we have allocated space on the stack for temporary variable. We will check this variable that it was set when a previous `NMI` is executing and clear if it is not nested `NMI`. We push `1` here to the previously allocated space on the stack to denote that a `non-maskable` interrupt executed currently. Remember that when and `NMI` or another exception occurs we have the following stack frame:

```
+------------------------+
|         SS             |
|         RSP            |
|        RFLAGS          |
|         CS             |
|         RIP            |
+------------------------+
```

and also an error code if an exception has it. So, after all of these manipulations our stack frame will look like this:

```
+-------------------------+
|          SS             |
|          RSP            |
|         RFLAGS          |
|          CS             |
|          RIP            |
|          RDX            |
|           1             |
+-------------------------+
```

In the next step we allocate yet another `40` bytes on the stack:

```
subq    $(5*8), %rsp
```

and pushes the copy of the original stack frame after the allocated space:

```
.rept 5
pushq     11*8(%rsp)
.endr
```

with the .rept assembly directive. We need in the copy of the original stack frame. Generally we need in two copies of the interrupt stack. First is `copied` interrupts stack: `saved` stack frame and `copied` stack frame. Now we pushes original stack frame to the `saved` stack frame which locates after the just allocated `40` bytes (`copied` stack frame). This stack frame is used to fixup the `copied` stack frame that a nested NMI may change. The second - `copied` stack frame modified by any nested `NMIs` to let the first `NMI` know that we triggered a second `NMI` and we should repeat the first `NMI` handler. Ok, we have made first copy of the original stack frame, now time to make second copy:

```
addq    $(10*8), %rsp

.rept 5
pushq     -6*8(%rsp)
.endr
subq    $(5*8), %rsp
```

After all of these manipulations our stack frame will be like this:

```
+-------------------------+
| original SS             |
| original Return RSP     |
| original RFLAGS         |
| original CS             |
| original RIP            |
+-------------------------+
| temp storage for rdx    |
+-------------------------+
| NMI executing variable  |
+-------------------------+
| copied SS               |
| copied Return RSP       |
| copied RFLAGS           |
| copied CS               |
| copied RIP              |
+-------------------------+
| Saved SS                |
| Saved Return RSP        |
| Saved RFLAGS            |
| Saved CS                |
| Saved RIP               |
+-------------------------+
```

After this we push dummy error code on the stack as we did it already in the previous exception handlers and allocate space for the general purpose registers on the stack:

```
pushq    $-1
ALLOC_PT_GPREGS_ON_STACK
```

We already saw implementation of the `ALLOC_PT_GREGS_ON_STACK` macro in the third part of the interrupts chapter. This macro defined in the arch/x86/entry/calling.h and yet another allocates `120` bytes on stack for the general purpose registers, from the `rdi` to the `r15` :

```
.macro ALLOC_PT_GPREGS_ON_STACK addskip=0
addq    $-(15*8+\addskip), %rsp
.endm
```

After space allocation for the general registers we can see call of the `paranoid_entry` :

```
call    paranoid_entry
```

We can remember from the previous parts this label. It pushes general purpose registers on the stack, reads `MSR_GS_BASE` Model Specific register and checks its value. If the value of the `MSR_GS_BASE` is negative, we came from the kernel mode and just return from the

`paranoid_entry`, in other way it means that we came from the usermode and need to execute `swapgs` instruction which will change user `gs` with the kernel `gs`:

```
ENTRY(paranoid_entry)
    cld
    SAVE_C_REGS 8
    SAVE_EXTRA_REGS 8
    movl    $1, %ebx
    movl    $MSR_GS_BASE, %ecx
    rdmsr
    testl    %edx, %edx
    js    1f
    SWAPGS
    xorl    %ebx, %ebx
1:    ret
END(paranoid_entry)
```

Note that after the `swapgs` instruction we zeroed the `ebx` register. Next time we will check content of this register and if we executed `swapgs` than `ebx` must contain `0` and `1` in other way. In the next step we store value of the `cr2` control register to the `r12` register, because the `NMI` handler can cause `page fault` and corrupt the value of this control register:

```
movq    %cr2, %r12
```

Now time to call actual `NMI` handler. We push the address of the `pt_regs` to the `rdi`, error code to the `rsi` and call the `do_nmi` handler:

```
movq    %rsp, %rdi
movq    $-1, %rsi
call    do_nmi
```

We will back to the `do_nmi` little later in this part, but now let's look what occurs after the `do_nmi` will finish its execution. After the `do_nmi` handler will be finished we check the `cr2` register, because we can got page fault during `do_nmi` performed and if we got it we restore original `cr2`, in other way we jump on the label `1`. After this we test content of the `ebx` register (remember it must contain `0` if we have used `swapgs` instruction and `1` if we didn't use it) and execute `SWAPGS_UNSAFE_STACK` if it contains `1` or jump to the `nmi_restore` label. The `SWAPGS_UNSAFE_STACK` macro just expands to the `swapgs` instruction. In the `nmi_restore` label we restore general purpose registers, clear allocated space on the stack for this registers, clear our temporary variable and exit from the interrupt handler with the `INTERRUPT_RETURN` macro:

```
    movq    %cr2, %rcx
    cmpq    %rcx, %r12
    je    1f
    movq    %r12, %cr2
1:
    testl    %ebx, %ebx
    jnz    nmi_restore
nmi_swapgs:
    SWAPGS_UNSAFE_STACK
nmi_restore:
    RESTORE_EXTRA_REGS
    RESTORE_C_REGS
    /* Pop the extra iret frame at once */
    REMOVE_PT_GPREGS_FROM_STACK 6*8
    /* Clear the NMI executing stack variable */
    movq    $0, 5*8(%rsp)
    INTERRUPT_RETURN
```

where `INTERRUPT_RETURN` is defined in the arch/x86/include/irqflags.h and just expands to the `iret` instruction. That's all.

Now let's consider case when another `NMI` interrupt occurred when previous `NMI` interrupt didn't finish its execution. You can remember from the beginning of this part that we've made a check that we came from userspace and jump on the `first_nmi` in this case:

```
cmpl    $__KERNEL_CS, 16(%rsp)
jne    first_nmi
```

Note that in this case it is first `NMI` every time, because if the first `NMI` catched page fault, breakpoint or another exception it will be executed in the kernel mode. If we didn't come from userspace, first of all we test our temporary variable:

```
cmpl    $1, -8(%rsp)
je    nested_nmi
```

and if it is set to `1` we jump to the `nested_nmi` label. If it is not `1` , we test the `IST` stack. In the case of nested `NMIs` we check that we are above the `repeat_nmi` . In this case we ignore it, in other way we check that we above than `end_repeat_nmi` and jump on the `nested_nmi_out` label.

Now let's look on the `do_nmi` exception handler. This function defined in the arch/x86/kernel/nmi.c source code file and takes two parameters:

- address of the `pt_regs` ;
- error code.

as all exception handlers. The `do_nmi` starts from the call of the `nmi_nesting_preprocess` function and ends with the call of the `nmi_nesting_postprocess`. The `nmi_nesting_preprocess` function checks that we likely do not work with the debug stack and if we on the debug stack set the `update_debug_stack` per-cpu variable to `1` and call the `debug_stack_set_zero` function from the arch/x86/kernel/cpu/common.c. This function increases the `debug_stack_use_ctr` per-cpu variable and loads new `Interrupt Descriptor Table`:

```
static inline void nmi_nesting_preprocess(struct pt_regs *regs)
{
        if (unlikely(is_debug_stack(regs->sp))) {
                debug_stack_set_zero();
                this_cpu_write(update_debug_stack, 1);
        }
}
```

The `nmi_nesting_postprocess` function checks the `update_debug_stack` per-cpu variable which we set in the `nmi_nesting_preprocess` and resets debug stack or in another words it loads origin `Interrupt Descriptor Table`. After the call of the `nmi_nesting_preprocess` function, we can see the call of the `nmi_enter` in the `do_nmi`. The `nmi_enter` increases `lockdep_recursion` field of the interrupted process, update preempt counter and informs the RCU subsystem about `NMI`. There is also `nmi_exit` function that does the same stuff as `nmi_enter`, but vice-versa. After the `nmi_enter` we increase `__nmi_count` in the `irq_stat` structure and call the `default_do_nmi` function. First of all in the `default_do_nmi` we check the address of the previous nmi and update address of the last nmi to the actual:

```
if (regs->ip == __this_cpu_read(last_nmi_rip))
    b2b = true;
else
    __this_cpu_write(swallow_nmi, false);

__this_cpu_write(last_nmi_rip, regs->ip);
```

After this first of all we need to handle CPU-specific `NMIs`:

```
handled = nmi_handle(NMI_LOCAL, regs, b2b);
__this_cpu_add(nmi_stats.normal, handled);
```

And then non-specific `NMIs` depends on its reason:

```C
reason = x86_platform.get_nmi_reason();
if (reason & NMI_REASON_MASK) {
    if (reason & NMI_REASON_SERR)
        pci_serr_error(reason, regs);
    else if (reason & NMI_REASON_IOCHK)
        io_check_error(reason, regs);

    __this_cpu_add(nmi_stats.external, 1);
    return;
}
```

That's all.

Range Exceeded Exception

The next exception is the `BOUND` range exceeded exception. The `BOUND` instruction determines if the first operand (array index) is within the bounds of an array specified the second operand (bounds operand). If the index is not within bounds, a `BOUND` range exceeded exception or `#BR` is occurred. The handler of the `#BR` exception is the `do_bounds` function that defined in the arch/x86/kernel/traps.c. The `do_bounds` handler starts with the call of the `exception_enter` function and ends with the call of the `exception_exit` :

```C
prev_state = exception_enter();

if (notify_die(DIE_TRAP, "bounds", regs, error_code,
               X86_TRAP_BR, SIGSEGV) == NOTIFY_STOP)
    goto exit;
...
...
...
exception_exit(prev_state);
return;
```

After we have got the state of the previous context, we add the exception to the `notify_die` chain and if it will return `NOTIFY_STOP` we return from the exception. More about notify chains and the `context tracking` functions you can read in the previous part. In the next step we enable interrupts if they were disabled with the `contidional_sti` function that checks `IF` flag and call the `local_irq_enable` depends on its value:

```C
conditional_sti(regs);

if (!user_mode(regs))
    die("bounds", regs, error_code);
```

and check that if we didn't came from user mode we send `SIGSEGV` signal with the `die` function. After this we check is MPX enabled or not, and if this feature is disabled we jump on the `exit_trap` label:

```C
if (!cpu_feature_enabled(X86_FEATURE_MPX)) {
    goto exit_trap;
}

where we execute `do_trap` function (more about it you can find in the previous part):

```C
exit_trap:
    do_trap(X86_TRAP_BR, SIGSEGV, "bounds", regs, error_code, NULL);
    exception_exit(prev_state);
```

If `MPX` feature is enabled we check the `BNDSTATUS` with the `get_xsave_field_ptr` function and if it is zero, it means that the `MPX` was not responsible for this exception:

```C
bndcsr = get_xsave_field_ptr(XSTATE_BNDCSR);
if (!bndcsr)
        goto exit_trap;
```

After all of this, there is still only one way when `MPX` is responsible for this exception. We will not dive into the details about Intel Memory Protection Extensions in this part, but will see it in another chapter.

Coprocessor exception and SIMD exception

The next two exceptions are x87 FPU Floating-Point Error exception or `#MF` and SIMD Floating-Point Exception or `#XF`. The first exception occurs when the `x87 FPU` has detected floating point error. For example divide by zero, numeric overflow and etc. The second exception occurs when the processor has detected SSE/SSE2/SSE3 `SIMD` floating-point exception. It can be the same as for the `x87 FPU`. The handlers for these exceptions are `do_coprocessor_error` and `do_simd_coprocessor_error` are defined in the arch/x86/kernel/traps.c and very similar on each other. They both make a call of the `math_error` function from the same source code file but pass different vector number. The `do_coprocessor_error` passes `X86_TRAP_MF` vector number to the `math_error`:
```

```C
dotraplinkage void do_coprocessor_error(struct pt_regs *regs, long error_code)
{
    enum ctx_state prev_state;

    prev_state = exception_enter();
    math_error(regs, error_code, X86_TRAP_MF);
    exception_exit(prev_state);
}
```

and `do_simd_coprocessor_error` passes `X86_TRAP_XF` to the `math_error` function:

```C
dotraplinkage void
do_simd_coprocessor_error(struct pt_regs *regs, long error_code)
{
    enum ctx_state prev_state;

    prev_state = exception_enter();
    math_error(regs, error_code, X86_TRAP_XF);
    exception_exit(prev_state);
}
```

First of all the `math_error` function defines current interrupted task, address of its fpu, string which describes an exception, add it to the `notify_die` chain and return from the exception handler if it will return `NOTIFY_STOP` :

```C
    struct task_struct *task = current;
    struct fpu *fpu = &task->thread.fpu;
    siginfo_t info;
    char *str = (trapnr == X86_TRAP_MF) ? "fpu exception" :
                        "simd exception";

    if (notify_die(DIE_TRAP, str, regs, error_code, trapnr, SIGFPE) == NOTIFY_STOP)
        return;
```

After this we check that we are from the kernel mode and if yes we will try to fix an exception with the `fixup_exception` function. If we cannot we fill the task with the exception's error code and vector number and die:

```C
if (!user_mode(regs)) {
    if (!fixup_exception(regs)) {
        task->thread.error_code = error_code;
        task->thread.trap_nr = trapnr;
        die(str, regs, error_code);
    }
    return;
}
```

If we came from the user mode, we save the `fpu` state, fill the task structure with the vector number of an exception and `siginfo_t` with the number of signal, `errno`, the address where exception occurred and signal code:

```
fpu__save(fpu);

task->thread.trap_nr    = trapnr;
task->thread.error_code = error_code;
info.si_signo        = SIGFPE;
info.si_errno        = 0;
info.si_addr        = (void __user *)uprobe_get_trap_addr(regs);
info.si_code = fpu__exception_code(fpu, trapnr);
```

After this we check the signal code and if it is non-zero we return:

```
if (!info.si_code)
    return;
```

Or send the `SIGFPE` signal in the end:

```
force_sig_info(SIGFPE, &info, task);
```

That's all.

Conclusion

It is the end of the sixth part of the Interrupts and Interrupt Handling chapter and we saw implementation of some exception handlers in this part, like `non-maskable` interrupt, SIMD and x87 FPU floating point exception. Finally we have finsihed with the `trap_init` function in this part and will go ahead in the next part. The next our point is the external interrupts and the `early_irq_init` function from the init/main.c.

If you have any questions or suggestions write me a comment or ping me at twitter.

Please note that English is not my first language, And I am really sorry for any inconvenience. If you find any mistakes please send me PR to linux-insides.

Links

- General Protection Fault
- opcode

* Non-Maskable
* BOUND instruction
* CPU socket
* Interrupt Descriptor Table
* Interrupt Stack Table
* Paravirtualization
* .rept
* SIMD
* Coprocessor
* x86_64
* iret
* page fault
* breakpoint
* Global Descriptor Table
* stack frame
* Model Specific regiser
* percpu
* RCU
* MPX
* x87 FPU
* Previous part

Interrupts and Interrupt Handling. Part 7.

Introduction to external interrupts

This is the seventh part of the Interrupts and Interrupt Handling in the Linux kernel chapter and in the previous part we have finished with the exceptions which are generated by the processor. In this part we will continue to dive to the interrupt handling and will start with the external hardware interrupt handling. As you can remember, in the previous part we have finished with the `trap_init` function from the arch/x86/kernel/trap.c and the next step is the call of the `early_irq_init` function from the init/main.c.

Interrupts are signal that are sent across IRQ or `Interrupt Request Line` by a hardware or software. External hardware interrupts allow devices like keyboard, mouse and etc, to indicate that it needs attention of the processor. Once the processor receives the `Interrupt Request`, it will temporary stop execution of the running program and invoke special routine which depends on an interrupt. We already know that this routine is called interrupt handler (or how we will call it `ISR` or `Interrupt Service Routine` from this part). The `ISR` or `Interrupt Handler Routine` can be found in Interrupt Vector table that is located at fixed address in the memory. After the interrupt is handled processor resumes the interrupted process. At the boot/initialization time, the Linux kernel identifies all devices in the machine, and appropriate interrupt handlers are loaded into the interrupt table. As we saw in the previous parts, most exceptions are handled simply by the sending a Unix signal to the interrupted process. That's why kernel is can handle an exception quickly. Unfortunately we can not use this approach for the external hardware interrupts, because often they arrive after (and sometimes long after) the process to which they are related has been suspended. So it would make no sense to send a Unix signal to the current process. External interrupt handling depends on the type of an interrupt:

- `I/O` interrupts;
- Timer interrupts;
- Interprocessor interrupts.

I will try to describe all types of interrupts in this book.

Generally, a handler of an `I/O` interrupt must be flexible enough to service several devices at the same time. For example in the PCI bus architecture several devices may share the same `IRQ` line. In the simplest way the Linux kernel must do following thing when an `I/O` interrupt occurred:

- Save the value of an `IRQ` and the register's contents on the kernel stack;

- Send an acknowledgment to the hardware controller which is servicing the `IRQ` line;
- Execute the interrupt service routine (next we will call it `ISR`) which is associated with the device;
- Restore registers and return from an interrupt;

Ok, we know a little theory and now let's start with the `early_irq_init` function. The implementation of the `early_irq_init` function is in the kernel/irq/irqdesc.c. This function make early initialization of the `irq_desc` structure. The `irq_desc` structure is the foundation of interrupt management code in the Linux kernel. An array of this structure, which has the same name - `irq_desc`, keeps track of every interrupt request source in the Linux kernel. This structure defined in the include/linux/irqdesc.h and as you can note it depends on the `CONFIG_SPARSE_IRQ` kernel configuration option. This kernel configuration option enables support for sparse irqs. The `irq_desc` structure contains many different files:

- `irq_common_data` - per irq and chip data passed down to chip functions;
- `status_use_accessors` - contains status of the interrupt source which is combination of the values from the `enum` from the include/linux/irq.h and different macros which are defined in the same source code file;
- `kstat_irqs` - irq stats per-cpu;
- `handle_irq` - highlevel irq-events handler;
- `action` - identifies the interrupt service routines to be invoked when the IRQ occurs;
- `irq_count` - counter of interrupt occurrences on the IRQ line;
- `depth` - `0` if the IRQ line is enabled and a positive value if it has been disabled at least once;
- `last_unhandled` - aging timer for unhandled count;
- `irqs_unhandled` - count of the unhandled interrupts;
- `lock` - a spin lock used to serialize the accesses to the `IRQ` descriptor;
- `pending_mask` - pending rebalanced interrupts;
- `owner` - an owner of interrupt descriptor. Interrupt descriptors can be allocated from modules. This field is need to proved refcount on the module which provides the interrupts;
- and etc.

Of course it is not all fields of the `irq_desc` structure, because it is too long to describe each field of this structure, but we will see it all soon. Now let's start to dive into the implementation of the `early_irq_init` function.

Early external interrupts initialization

Now, let's look on the implementation of the `early_irq_init` function. Note that implementation of the `early_irq_init` function depends on the `CONFIG_SPARSE_IRQ` kernel configuration option. Now we consider implementation of the `early_irq_init` function when the `CONFIG_SPARSE_IRQ` kernel configuration option is not set. This function starts from the declaration of the following variables: `irq` descriptors counter, loop counter, memory node and the `irq_desc` descriptor:

```
int __init early_irq_init(void)
{
        int count, i, node = first_online_node;
        struct irq_desc *desc;
        ...
        ...
        ...
}
```

The `node` is an online NUMA node which depends on the `MAX_NUMNODES` value which depends on the `CONFIG_NODES_SHIFT` kernel configuration parameter:

```
#define MAX_NUMNODES    (1 << NODES_SHIFT)
...
...
...
#ifdef CONFIG_NODES_SHIFT
    #define NODES_SHIFT     CONFIG_NODES_SHIFT
#else
    #define NODES_SHIFT     0
#endif
```

As I already wrote, implementation of the `first_online_node` macro depends on the `MAX_NUMNODES` value:

```
#if MAX_NUMNODES > 1
  #define first_online_node       first_node(node_states[N_ONLINE])
#else
  #define first_online_node       0
```

The `node_states` is the enum which defined in the include/linux/nodemask.h and represent the set of the states of a node. In our case we are searching an online node and it will be `0` if `MAX_NUMNODES` is one or zero. If the `MAX_NUMNODES` is greater than one, the `node_states[N_ONLINE]` will return `1` and the `first_node` macro will be expands to the call of the `__first_node` function which will return `minimal` or the first online node:

```
#define first_node(src) __first_node(&(src))

static inline int __first_node(const nodemask_t *srcp)
{
        return min_t(int, MAX_NUMNODES, find_first_bit(srcp->bits, MAX_NUMNODES));
}
```

More about this will be in the another chapter about the `NUMA` . The next step after the declaration of these local variables is the call of the:

```
init_irq_default_affinity();
```

function. The `init_irq_default_affinity` function defined in the same source code file and depends on the `CONFIG_SMP` kernel configuration option allocates a given cpumask structure (in our case it is the `irq_default_affinity`):

```
#if defined(CONFIG_SMP)
cpumask_var_t irq_default_affinity;

static void __init init_irq_default_affinity(void)
{
        alloc_cpumask_var(&irq_default_affinity, GFP_NOWAIT);
        cpumask_setall(irq_default_affinity);
}
#else
static void __init init_irq_default_affinity(void)
{
}
#endif
```

We know that when a hardware, such as disk controller or keyboard, needs attention from the processor, it throws an interrupt. The interrupt tells to the processor that something has happened and that the processor should interrupt current process and handle an incoming event. In order to prevent multiple devices from sending the same interrupts, the IRQ system was established where each device in a computer system is assigned its own special IRQ so that its interrupts are unique. Linux kernel can assign certain `IRQs` to specific processors. This is known as `SMP IRQ affinity` , and it allows you control how your system will respond to various hardware events (that's why it has certain implementation only if the `CONFIG_SMP` kernel configuration option is set). After we allocated `irq_default_affinity` cpumask, we can see `printk` output:

```
printk(KERN_INFO "NR_IRQS:%d\n", NR_IRQS);
```

which prints `NR_IRQS` :

```
~$ dmesg | grep NR_IRQS
[    0.000000] NR_IRQS:4352
```

The `NR_IRQS` is the maximum number of the `irq` descriptors or in another words maximum number of interrupts. Its value depends on the state of the `CONFIG_X86_IO_APIC` kernel configuration option. If the `CONFIG_X86_IO_APIC` is not set and the Linux kernel uses an old PIC chip, the `NR_IRQS` is:

```
#define NR_IRQS_LEGACY                    16

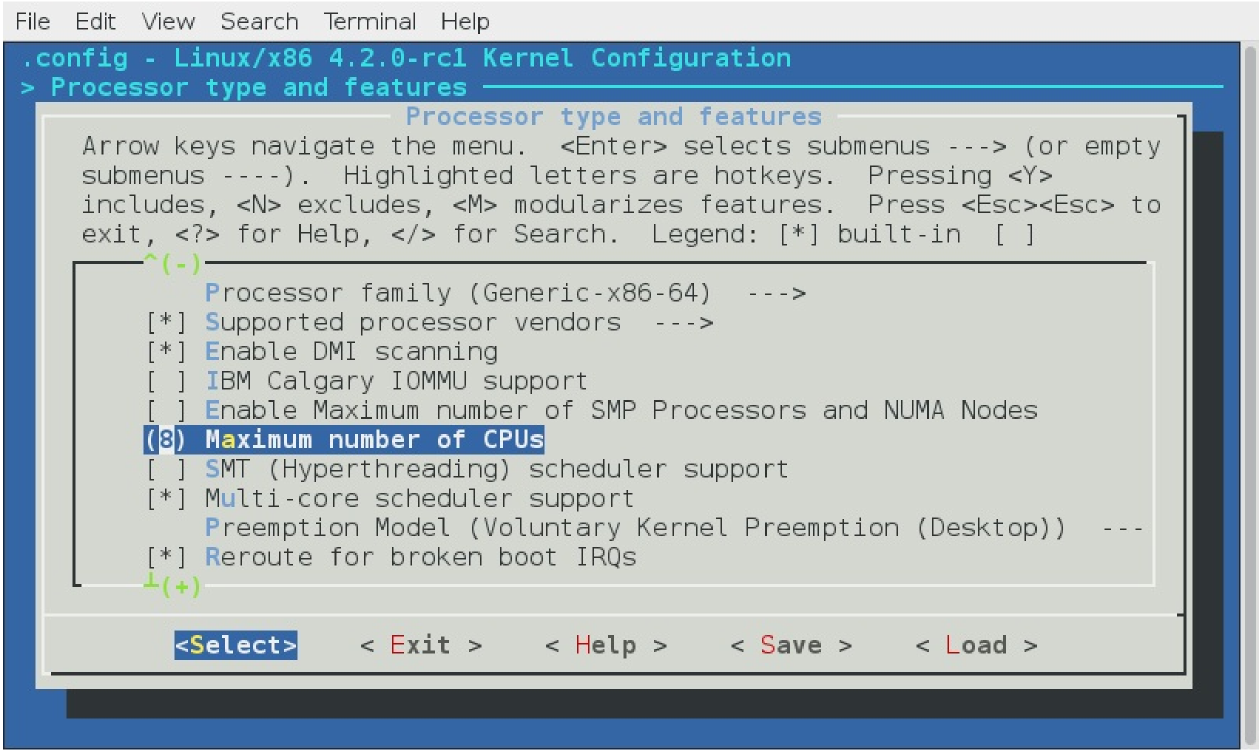
#ifdef CONFIG_X86_IO_APIC
...
...
...
#else
# define NR_IRQS                        NR_IRQS_LEGACY
#endif
```

In other way, when the `CONFIG_X86_IO_APIC` kernel configuration option is set, the `NR_IRQS` depends on the amount of the processors and amount of the interrupt vectors:

```
#define CPU_VECTOR_LIMIT              (64 * NR_CPUS)
#define NR_VECTORS                    256
#define IO_APIC_VECTOR_LIMIT          ( 32 * MAX_IO_APICS )
#define MAX_IO_APICS                  128

# define NR_IRQS                                        \
        (CPU_VECTOR_LIMIT > IO_APIC_VECTOR_LIMIT ?      \
                (NR_VECTORS + CPU_VECTOR_LIMIT)  :      \
                (NR_VECTORS + IO_APIC_VECTOR_LIMIT))
...
...
...
```

We remember from the previous parts, that the amount of processors we can set during Linux kernel configuration process with the `CONFIG_NR_CPUS` configuration option:



In the first case (`CPU_VECTOR_LIMIT > IO_APIC_VECTOR_LIMIT`), the `NR_IRQS` will be `4352` , in the second case (`CPU_VECTOR_LIMIT < IO_APIC_VECTOR_LIMIT`), the `NR_IRQS` will be `768` . In my case the `NR_CPUS` is `8` as you can see in the my configuration, the `CPU_VECTOR_LIMIT` is `512` and the `IO_APIC_VECTOR_LIMIT` is `4096` . So `NR_IRQS` for my configuration is `4352` :

```
~$ dmesg | grep NR_IRQS
[    0.000000] NR_IRQS:4352
```

In the next step we assign array of the IRQ descriptors to the `irq_desc` variable which we defined in the start of the `early_irq_init` function and calculate count of the `irq_desc` array with the `ARRAY_SIZE` macro:

```
desc = irq_desc;
count = ARRAY_SIZE(irq_desc);
```

The `irq_desc` array defined in the same source code file and looks like:

```
struct irq_desc irq_desc[NR_IRQS] __cacheline_aligned_in_smp = {
        [0 ... NR_IRQS-1] = {
                .handle_irq     = handle_bad_irq,
                .depth          = 1,
                .lock           = __RAW_SPIN_LOCK_UNLOCKED(irq_desc->lock),
        }
};
```

The `irq_desc` is array of the `irq` descriptors. It has three already initialized fields:

- `handle_irq` - as I already wrote above, this field is the highlevel irq-event handler. In our case it initialized with the `handle_bad_irq` function that defined in the kernel/irq/handle.c source code file and handles spurious and unhandled irqs;
- `depth` - `0` if the IRQ line is enabled and a positive value if it has been disabled at least once;
- `lock` - A spin lock used to serialize the accesses to the `IRQ` descriptor.

As we calculated count of the interrupts and initialized our `irq_desc` array, we start to fill descriptors in the loop:

```
for (i = 0; i < count; i++) {
    desc[i].kstat_irqs = alloc_percpu(unsigned int);
    alloc_masks(&desc[i], GFP_KERNEL, node);
    raw_spin_lock_init(&desc[i].lock);
    lockdep_set_class(&desc[i].lock, &irq_desc_lock_class);
    desc_set_defaults(i, &desc[i], node, NULL);
}
```

We are going through the all interrupt descriptors and do the following things:

First of all we allocate percpu variable for the `irq` kernel statistic with the `alloc_percpu` macro. This macro allocates one instance of an object of the given type for every processor on the system. You can access kernel statistic from the userspace via `/proc/stat` :

```
~$ cat /proc/stat
cpu  207907 68 53904 5427850 14394 0 394 0 0 0
cpu0 25881 11 6684 679131 1351 0 18 0 0 0
cpu1 24791 16 5894 679994 2285 0 24 0 0 0
cpu2 26321 4 7154 678924 664 0 71 0 0 0
cpu3 26648 8 6931 678891 414 0 244 0 0 0
...
...
...
```

Where the sixth column is the servicing interrupts. After this we allocate cpumask for the given irq descriptor affinity and initialize the spinlock for the given interrupt descriptor. After this before the critical section, the lock will be acquired with a call of the `raw_spin_lock` and unlocked with the call of the `raw_spin_unlock` . In the next step we call the `lockdep_set_class` macro which set the Lock validator `irq_desc_lock_class` class for the lock of the given interrupt descriptor. More about `lockdep` , `spinlock` and other synchronization primitives will be described in the separate chapter.

In the end of the loop we call the `desc_set_defaults` function from the kernel/irq/irqdesc.c. This function takes four parameters:

- number of a irq;
- interrupt descriptor;
- online `NUMA` node;
- owner of interrupt descriptor. Interrupt descriptors can be allocated from modules. This field is need to proved refcount on the module which provides the interrupts;

and fills the rest of the `irq_desc` fields. The `desc_set_defaults` function fills interrupt number, `irq` chip, platform-specific per-chip private data for the chip methods, per-IRQ data for the `irq_chip` methods and MSI descriptor for the per `irq` and `irq` chip data:

```
desc->irq_data.irq = irq;
desc->irq_data.chip = &no_irq_chip;
desc->irq_data.chip_data = NULL;
desc->irq_data.handler_data = NULL;
desc->irq_data.msi_desc = NULL;
...
...
...
```

The `irq_data.chip` structure provides general `API` like the `irq_set_chip`, `irq_set_irq_type` and etc, for the irq controller drivers. You can find it in the kernel/irq/chip.c source code file.

After this we set the status of the accessor for the given descriptor and set disabled state of the interrupts:

```
...
...
...
irq_settings_clr_and_set(desc, ~0, _IRQ_DEFAULT_INIT_FLAGS);
irqd_set(&desc->irq_data, IRQD_IRQ_DISABLED);
...
...
...
```

In the next step we set the high level interrupt handlers to the `handle_bad_irq` which handles spurious and unhandled irqs (as the hardware stuff is not initialized yet, we set this handler), set `irq_desc.desc` to `1` which means that an `IRQ` is disabled, reset count of the unhandled interrupts and interrupts in general:

```
...
...
...
desc->handle_irq = handle_bad_irq;
desc->depth = 1;
desc->irq_count = 0;
desc->irqs_unhandled = 0;
desc->name = NULL;
desc->owner = owner;
...
...
...
```

After this we go through the all possible processor with the for_each_possible_cpu helper and set the `kstat_irqs` to zero for the given interrupt descriptor:

```
    for_each_possible_cpu(cpu)
        *per_cpu_ptr(desc->kstat_irqs, cpu) = 0;
```

and call the `desc_smp_init` function from the kernel/irq/irqdesc.c that initializes `NUMA` node of the given interrupt descriptor, sets default `SMP` affinity and clears the `pending_mask` of the given interrupt descriptor depends on the value of the `CONFIG_GENERIC_PENDING_IRQ` kernel configuration option:

```
static void desc_smp_init(struct irq_desc *desc, int node)
{
        desc->irq_data.node = node;
        cpumask_copy(desc->irq_data.affinity, irq_default_affinity);
#ifdef CONFIG_GENERIC_PENDING_IRQ
        cpumask_clear(desc->pending_mask);
#endif
}
```

In the end of the `early_irq_init` function we return the return value of the `arch_early_irq_init` function:

```
return arch_early_irq_init();
```

This function defined in the kernel/apic/vector.c and contains only one call of the `arch_early_ioapic_init` function from the kernel/apic/io_apic.c. As we can understand from the `arch_early_ioapic_init` function's name, this function makes early initialization of the I/O APIC. First of all it make a check of the number of the legacy interrupts with the call of the `nr_legacy_irqs` function. If we have no legacy interrupts with the Intel 8259 programmable interrupt controller we set `io_apic_irqs` to the `0xffffffffffffffff`:

```
if (!nr_legacy_irqs())
    io_apic_irqs = ~0UL;
```

After this we are going through the all `I/O APICs` and allocate space for the registers with the call of the `alloc_ioapic_saved_registers` :

```
for_each_ioapic(i)
    alloc_ioapic_saved_registers(i);
```

And in the end of the `arch_early_ioapic_init` function we are going through the all legacy irqs (from `IRQ0` to `IRQ15`) in the loop and allocate space for the `irq_cfg` which represents configuration of an irq on the given `NUMA` node:

```
for (i = 0; i < nr_legacy_irqs(); i++) {
    cfg = alloc_irq_and_cfg_at(i, node);
    cfg->vector = IRQ0_VECTOR + i;
    cpumask_setall(cfg->domain);
}
```

That's all.

Sparse IRQs

We already saw in the beginning of this part that implementation of the `early_irq_init` function depends on the `CONFIG_SPARSE_IRQ` kernel configuration option. Previously we saw implementation of the `early_irq_init` function when the `CONFIG_SPARSE_IRQ` configuration option is not set, now let's look on the its implementation when this option is set. Implementation of this function very similar, but little differ. We can see the same definition of variables and call of the `init_irq_default_affinity` in the beginning of the `early_irq_init` function:

```
#ifdef CONFIG_SPARSE_IRQ
int __init early_irq_init(void)
{
	int i, initcnt, node = first_online_node;
	struct irq_desc *desc;

	init_irq_default_affinity();
	...
	...
	...
}
#else
...
...
...
```

But after this we can see the following call:

```
initcnt = arch_probe_nr_irqs();
```

The `arch_probe_nr_irqs` function defined in the arch/x86/kernel/apic/vector.c and calculates count of the pre-allocated irqs and update `nr_irqs` with its number. But stop. Why there are pre-allocated irqs? There is alternative form of interrupts called - Message Signaled Interrupts available in the PCI. Instead of assigning a fixed number of the interrupt request, the device is allowed to record a message at a particular address of RAM, in fact, the display on the Local APIC. `MSI` permits a device to allocate `1`, `2`, `4`, `8`, `16` or `32` interrupts and `MSI-X` permits a device to allocate up to `2048` interrupts. Now we know that irqs can be pre-allocated. More about `MSI` will be in a next part, but now let's look on the `arch_probe_nr_irqs` function. We can see the check which assign amount of the interrupt vectors for the each processor in the system to the `nr_irqs` if it is greater and calculate the `nr` which represents number of `MSI` interrupts:

```
int nr_irqs = NR_IRQS;

if (nr_irqs > (NR_VECTORS * nr_cpu_ids))
	nr_irqs = NR_VECTORS * nr_cpu_ids;

nr = (gsi_top + nr_legacy_irqs()) + 8 * nr_cpu_ids;
```

Take a look on the `gsi_top` variable. Each `APIC` is identified with its own `ID` and with the offset where its `IRQ` starts. It is called `GSI` base or `Global System Interrupt` base. So the `gsi_top` represents it. We get the `Global System Interrupt` base from the MultiProcessor Configuration Table table (you can remember that we have parsed this table in the sixth part of the Linux Kernel initialization process chapter).

After this we update the `nr` depends on the value of the `gsi_top`:

```
#if defined(CONFIG_PCI_MSI) || defined(CONFIG_HT_IRQ)
        if (gsi_top <= NR_IRQS_LEGACY)
                nr +=  8 * nr_cpu_ids;
        else
                nr += gsi_top * 16;
#endif
```

Update the `nr_irqs` if it less than `nr` and return the number of the legacy irqs:

```
if (nr < nr_irqs)
    nr_irqs = nr;

return nr_legacy_irqs();
}
```

The next after the `arch_probe_nr_irqs` is printing information about number of `IRQs`:

```
printk(KERN_INFO "NR_IRQS:%d nr_irqs:%d %d\n", NR_IRQS, nr_irqs, initcnt);
```

We can find it in the dmesg output:

```
$ dmesg | grep NR_IRQS
[    0.000000] NR_IRQS:4352 nr_irqs:488 16
```

After this we do some checks that `nr_irqs` and `initcnt` values is not greater than maximum allowable number of `irqs`:

```
if (WARN_ON(nr_irqs > IRQ_BITMAP_BITS))
    nr_irqs = IRQ_BITMAP_BITS;

if (WARN_ON(initcnt > IRQ_BITMAP_BITS))
    initcnt = IRQ_BITMAP_BITS;
```

where `IRQ_BITMAP_BITS` is equal to the `NR_IRQS` if the `CONFIG_SPARSE_IRQ` is not set and `NR_IRQS + 8196` in other way. In the next step we are going over all interrupt descriptors which need to be allocated in the loop and allocate space for the descriptor and insert to the `irq_desc_tree` radix tree:

```
for (i = 0; i < initcnt; i++) {
    desc = alloc_desc(i, node, NULL);
    set_bit(i, allocated_irqs);
    irq_insert_desc(i, desc);
}
```

In the end of the `early_irq_init` function we return the value of the call of the `arch_early_irq_init` function as we did it already in the previous variant when the `CONFIG_SPARSE_IRQ` option was not set:

```
return arch_early_irq_init();
```

That's all.

Conclusion

It is the end of the seventh part of the Interrupts and Interrupt Handling chapter and we started to dive into external hardware interrupts in this part. We saw early initialization of the `irq_desc` structure which represents description of an external interrupt and contains information about it like list of irq actions, information about interrupt handler, interrupt's owner, count of the unhandled interrupt and etc. In the next part we will continue to research external interrupts.

If you have any questions or suggestions write me a comment or ping me at twitter.

Please note that English is not my first language, And I am really sorry for any inconvenience. If you find any mistakes please send me PR to linux-insides.

Links

- IRQ
- numa
- Enum type
- cpumask
- percpu
- spinlock
- critical section
- Lock validator
- MSI
- I/O APIC

- Local APIC
- Intel 8259
- PIC
- MultiProcessor Configuration Table
- radix tree
- dmesg

Interrupts and Interrupt Handling. Part 8.

Non-early initialization of the IRQs

This is the eighth part of the Interrupts and Interrupt Handling in the Linux kernel chapter and in the previous part we started to dive into the external hardware interrupts. We looked on the implementation of the `early_irq_init` function from the kernel/irq/irqdesc.c source code file and saw the initialization of the `irq_desc` structure in this function. Remind that `irq_desc` structure (defined in the include/linux/irqdesc.h is the foundation of interrupt management code in the Linux kernel and represents an interrupt descriptor. In this part we will continue to dive into the initialization stuff which is related to the external hardware interrupts.

Right after the call of the `early_irq_init` function in the init/main.c we can see the call of the `init_IRQ` function. This function is architecture-specific and defined in the arch/x86/kernel/irqinit.c. The `init_IRQ` function makes initialization of the `vector_irq` percpu variable that defined in the same arch/x86/kernel/irqinit.c source code file:

```
...
DEFINE_PER_CPU(vector_irq_t, vector_irq) = {
         [0 ... NR_VECTORS - 1] = -1,
};
...
```

and represents `percpu` array of the interrupt vector numbers. The `vector_irq_t` defined in the arch/x86/include/asm/hw_irq.h and expands to the:

```
typedef int vector_irq_t[NR_VECTORS];
```

where `NR_VECTORS` is count of the vector number and as you can remember from the first part of this chapter it is `256` for the x86_64:

```
#define NR_VECTORS                         256
```

So, in the start of the `init_IRQ` function we fill the `vector_irq` percpu array with the vector number of the `legacy` interrupts:

```
void __init init_IRQ(void)
{
    int i;

    for (i = 0; i < nr_legacy_irqs(); i++)
        per_cpu(vector_irq, 0)[IRQ0_VECTOR + i] = i;
...
...
...
}
```

This `vector_irq` will be used during the first steps of an external hardware interrupt handling in the `do_IRQ` function from the arch/x86/kernel/irq.c:

```
__visible unsigned int __irq_entry do_IRQ(struct pt_regs *regs)
{
    ...
    ...
    ...
    irq = __this_cpu_read(vector_irq[vector]);

    if (!handle_irq(irq, regs)) {
        ...
        ...
        ...
    }

    exiting_irq();
    ...
    ...
    return 1;
}
```

Why is `legacy` here? Actually all interrupts are handled by the modern IO-APIC controller. But these interrupts (from `0x30` to `0x3f`) by legacy interrupt-controllers like Programmable Interrupt Controller. If these interrupts are handled by the `I/O APIC` then this vector space will be freed and re-used. Let's look on this code closer. First of all the `nr_legacy_irqs` defined in the arch/x86/include/asm/i8259.h and just returns the `nr_legacy_irqs` field from the `legacy_pic` structure:

```
static inline int nr_legacy_irqs(void)
{
        return legacy_pic->nr_legacy_irqs;
}
```

This structure defined in the same header file and represents non-modern programmable interrupts controller:

```
struct legacy_pic {
        int nr_legacy_irqs;
        struct irq_chip *chip;
        void (*mask)(unsigned int irq);
        void (*unmask)(unsigned int irq);
        void (*mask_all)(void);
        void (*restore_mask)(void);
        void (*init)(int auto_eoi);
        int (*irq_pending)(unsigned int irq);
        void (*make_irq)(unsigned int irq);
};
```

Actual default maximum number of the legacy interrupts represented by the `NR_IRQ_LEGACY` macro from the arch/x86/include/asm/irq_vectors.h:

```
#define NR_IRQS_LEGACY                    16
```

In the loop we are accessing the `vecto_irq` per-cpu array with the `per_cpu` macro by the `IRQ0_VECTOR + i` index and write the legacy vector number there. The `IRQ0_VECTOR` macro defined in the arch/x86/include/asm/irq_vectors.h header file and expands to the `0x30` :

```
#define FIRST_EXTERNAL_VECTOR           0x20

#define IRQ0_VECTOR                     ((FIRST_EXTERNAL_VECTOR + 16) & ~15)
```

Why is `0x30` here? You can remember from the first part of this chapter that first 32 vector numbers from `0` to `31` are reserved by the processor and used for the processing of architecture-defined exceptions and interrupts. Vector numbers from `0x30` to `0x3f` are reserved for the ISA. So, it means that we fill the `vector_irq` from the `IRQ0_VECTOR` which is equal to the `32` to the `IRQ0_VECTOR + 16` (before the `0x30`).

In the end of the `init_IRQ` function we can see the call of the following function:

```
x86_init.irqs.intr_init();
```

from the arch/x86/kernel/x86_init.c source code file. If you have read chapter about the Linux kernel initialization process, you can remember the `x86_init` structure. This structure contains a couple of files which are points to the function related to the platform setup (`x86_64` in our case), for example `resources` - related with the memory resources,

`mpparse` - related with the parsing of the MultiProcessor Configuration Table table and etc.). As we can see the `x86_init` also contains the `irqs` field which contains three following fields:

```
struct x86_init_ops x86_init __initdata
{
    ...
    ...
    ...
    .irqs = {
                .pre_vector_init        = init_ISA_irqs,
                .intr_init              = native_init_IRQ,
                .trap_init              = x86_init_noop,
    },
    ...
    ...
    ...
}
```

Now, we are interesting in the `native_init_IRQ`. As we can note, the name of the `native_init_IRQ` function contains the `native_` prefix which means that this function is architecture-specific. It defined in the arch/x86/kernel/irqinit.c and executes general initialization of the Local APIC and initialization of the ISA irqs. Let's look on the implementation of the `native_init_IRQ` function and will try to understand what occurs there. The `native_init_IRQ` function starts from the execution of the following function:

```
x86_init.irqs.pre_vector_init();
```

As we can see above, the `pre_vector_init` points to the `init_ISA_irqs` function that defined in the same source code file and as we can understand from the function's name, it makes initialization of the `ISA` related interrupts. The `init_ISA_irqs` function starts from the definition of the `chip` variable which has a `irq_chip` type:

```
void __init init_ISA_irqs(void)
{
    struct irq_chip *chip = legacy_pic->chip;
    ...
    ...
    ...
```

The `irq_chip` structure defined in the include/linux/irq.h header file and represents hardware interrupt chip descriptor. It contains:

- `name` - name of a device. Used in the `/proc/interrupts`:

```
$ cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6
  CPU7
  0:         16          0          0          0          0          0          0
     0   IO-APIC    2-edge      timer
  1:          2          0          0          0          0          0          0
     0   IO-APIC    1-edge      i8042
  8:          1          0          0          0          0          0          0
     0   IO-APIC    8-edge      rtc0
```

look on the last column;

- `(*irq_mask)(struct irq_data *data)` - mask an interrupt source;
- `(*irq_ack)(struct irq_data *data)` - start of a new interrupt;
- `(*irq_startup)(struct irq_data *data)` - start up the interrupt;
- `(*irq_shutdown)(struct irq_data *data)` - shutdown the interrupt
- and etc.

fields. Note that the `irq_data` structure represents set of the per irq chip data passed down to chip functions. It contains `mask` - precomputed bitmask for accessing the chip registers, `irq` - interrupt number, `hwirq` - hardware interrupt number, local to the interrupt domain chip low level interrupt hardware access and etc.

After this depends on the `CONFIG_X86_64` and `CONFIG_X86_LOCAL_APIC` kernel configuration option call the `init_bsp_APIC` function from the arch/x86/kernel/apic/apic.c:

```
#if defined(CONFIG_X86_64) || defined(CONFIG_X86_LOCAL_APIC)
    init_bsp_APIC();
#endif
```

This function makes initialization of the APIC of `bootstrap processor` (or processor which starts first). It starts from the check that we found SMP config (read more about it in the sixth part of the Linux kernel initialization process chapter) and the processor has `APIC` :

```
if (smp_found_config || !cpu_has_apic)
    return;
```

In other way we return from this function. In the next step we call the `clear_local_APIC` function from the same source code file that shutdowns the local `APIC` (more about it will be in the chapter about the `Advanced Programmable Interrupt Controller`) and enable `APIC` of the first processor by the setting `unsigned int value` to the `APIC_SPIV_APIC_ENABLED` :

```
value = apic_read(APIC_SPIV);
value &= ~APIC_VECTOR_MASK;
value |= APIC_SPIV_APIC_ENABLED;
```

and writing it with the help of the `apic_write` function:

```
apic_write(APIC_SPIV, value);
```

After we have enabled `APIC` for the bootstrap processor, we return to the `init_ISA_irqs` function and in the next step we initialize legacy `Programmable Interrupt Controller` and set the legacy chip and handler for the each legacy irq:

```
legacy_pic->init(0);

for (i = 0; i < nr_legacy_irqs(); i++)
    irq_set_chip_and_handler(i, chip, handle_level_irq);
```

Where can we find `init` function? The `legacy_pic` defined in the arch/x86/kernel/i8259.c and it is:

```
struct legacy_pic *legacy_pic = &default_legacy_pic;
```

Where the `default_legacy_pic` is:

```
struct legacy_pic default_legacy_pic = {
    ...
    ...
    ...
    .init = init_8259A,
    ...
    ...
    ...
}
```

The `init_8259A` function defined in the same source code file and executes initialization of the Intel 8259 ``Programmable Interrupt Controller` (more about it will be in the separate chapter about `Programmable Interrupt Controllers` and `APIC`).

Now we can return to the `native_init_IRQ` function, after the `init_ISA_irqs` function finished its work. The next step is the call of the `apic_intr_init` function that allocates special interrupt gates which are used by the SMP architecture for the Inter-processor interrupt. The `alloc_intr_gate` macro from the arch/x86/include/asm/desc.h used for the interrupt descriptor allocation:

```
#define alloc_intr_gate(n, addr)                        \
do {                                                    \
        alloc_system_vector(n);                         \
        set_intr_gate(n, addr);                         \
} while (0)
```

As we can see, first of all it expands to the call of the `alloc_system_vector` function that checks the given vector number in the `used_vectors` bitmap (read previous part about it) and if it is not set in the `used_vectors` bitmap we set it. After this we test that the `first_system_vector` is greater than given interrupt vector number and if it is greater we assign it:

```
if (!test_bit(vector, used_vectors)) {
    set_bit(vector, used_vectors);
    if (first_system_vector > vector)
        first_system_vector = vector;
} else {
    BUG();
}
```

We already saw the `set_bit` macro, now let's look on the `test_bit` and the `first_system_vector`. The first `test_bit` macro defined in the arch/x86/include/asm/bitops.h and looks like this:

```
#define test_bit(nr, addr)                      \
        (__builtin_constant_p((nr))             \
         ? constant_test_bit((nr), (addr))      \
         : variable_test_bit((nr), (addr)))
```

We can see the ternary operator here make a test with the gcc built-in function `__builtin_constant_p` tests that given vector number (`nr`) is known at compile time. If you're feeling misunderstanding of the `__builtin_constant_p` , we can make simple test:

```
#include <stdio.h>

#define PREDEFINED_VAL 1

int main() {
    int i = 5;
    printf("__builtin_constant_p(i) is %d\n", __builtin_constant_p(i));
    printf("__builtin_constant_p(PREDEFINED_VAL) is %d\n", __builtin_constant_p(PREDEF
INED_VAL));
    printf("__builtin_constant_p(100) is %d\n", __builtin_constant_p(100));

    return 0;
}
```

and look on the result:

```
$ gcc test.c -o test
$ ./test
__builtin_constant_p(i) is 0
__builtin_constant_p(PREDEFINED_VAL) is 1
__builtin_constant_p(100) is 1
```

Now I think it must be clear for you. Let's get back to the `test_bit` macro. If the `__builtin_constant_p` will return non-zero, we call `constant_test_bit` function:

```
static inline int constant_test_bit(int nr, const void *addr)
{
    const u32 *p = (const u32 *)addr;

    return ((1UL << (nr & 31)) & (p[nr >> 5])) != 0;
}
```

and the `variable_test_bit` in other way:

```
static inline int variable_test_bit(int nr, const void *addr)
{
        u8 v;
        const u32 *p = (const u32 *)addr;

        asm("btl %2,%1; setc %0" : "=qm" (v) : "m" (*p), "Ir" (nr));
        return v;
}
```

What's the difference between two these functions and why do we need in two different functions for the same purpose? As you already can guess main purpose is optimization. If we will write simple example with these functions:

```
#define CONST 25

int main() {
    int nr = 24;
    variable_test_bit(nr, (int*)0x10000000);
    constant_test_bit(CONST, (int*)0x10000000)
    return 0;
}
```

and will look on the assembly output of our example we will see following assembly code:

```
pushq    %rbp
movq    %rsp, %rbp

movl    $268435456, %esi
movl    $25, %edi
call    constant_test_bit
```

for the `constant_test_bit` , and:

```
pushq    %rbp
movq    %rsp, %rbp

subq    $16, %rsp
movl    $24, -4(%rbp)
movl    -4(%rbp), %eax
movl    $268435456, %esi
movl    %eax, %edi
call    variable_test_bit
```

for the `variable_test_bit` . These two code listings starts with the same part, first of all we save base of the current stack frame in the `%rbp` register. But after this code for both examples is different. In the first example we put `$268435456` (here the `$268435456` is our second parameter - `0x10000000`) to the `esi` and `$25` (our first parameter) to the `edi` register and call `constant_test_bit` . We put function parameters to the `esi` and `edi` registers because as we are learning Linux kernel for the `x86_64` architecture we use `System V AMD64 ABI` calling convention. All is pretty simple. When we are using predefined constant, the compiler can just substitute its value. Now let's look on the second part. As you can see here, the compiler can not substitute value from the `nr` variable. In this case compiler must calculate its offset on the program's stack frame. We subtract `16` from the `rsp` register to allocate stack for the local variables data and put the `$24` (value of the `nr` variable) to the `rbp` with offset `-4` . Our stack frame will be like this:

```
         <- stack grows

              %[rbp]
                 |
+----------+ +---------+ +---------+ +--------+
|          | |         | | return  | |        |
|    nr    |-|         |-|         |-|  argc  |
|          | |         | | address | |        |
+----------+ +---------+ +---------+ +--------+
                 |
              %[rsp]
```

After this we put this value to the `eax`, so `eax` register now contains value of the `nr`. In the end we do the same that in the first example, we put the `$268435456` (the first parameter of the `variable_test_bit` function) and the value of the `eax` (value of `nr`) to the `edi` register (the second parameter of the `variable_test_bit function`).

The next step after the `apic_intr_init` function will finish its work is the setting interrupt gates from the `FIRST_EXTERNAL_VECTOR` or `0x20` to the `0x256`:

```
i = FIRST_EXTERNAL_VECTOR;

#ifndef CONFIG_X86_LOCAL_APIC
#define first_system_vector NR_VECTORS
#endif

for_each_clear_bit_from(i, used_vectors, first_system_vector) {
    set_intr_gate(i, irq_entries_start + 8 * (i - FIRST_EXTERNAL_VECTOR));
}
```

But as we are using the `for_each_clear_bit_from` helper, we set only non-initialized interrupt gates. After this we use the same `for_each_clear_bit_from` helper to fill the non-filled interrupt gates in the interrupt table with the `spurious_interrupt`:

```
#ifdef CONFIG_X86_LOCAL_APIC
for_each_clear_bit_from(i, used_vectors, NR_VECTORS)
    set_intr_gate(i, spurious_interrupt);
#endif
```

Where the `spurious_interrupt` function represent interrupt handler for the `spurious` interrupt. Here the `used_vectors` is the `unsigned long` that contains already initialized interrupt gates. We already filled first `32` interrupt vectors in the `trap_init` function from the arch/x86/kernel/setup.c source code file:

```
for (i = 0; i < FIRST_EXTERNAL_VECTOR; i++)
    set_bit(i, used_vectors);
```

You can remember how we did it in the sixth part of this chapter.

In the end of the `native_init_IRQ` function we can see the following check:

```
if (!acpi_ioapic && !of_ioapic && nr_legacy_irqs())
    setup_irq(2, &irq2);
```

First of all let's deal with the condition. The `acpi_ioapic` variable represents existence of I/O APIC. It defined in the arch/x86/kernel/acpi/boot.c. This variable set in the `acpi_set_irq_model_ioapic` function that called during the processing `Multiple APIC Description Table`. This occurs during initialization of the architecture-specific stuff in the arch/x86/kernel/setup.c (more about it we will know in the other chapter about APIC). Note that the value of the `acpi_ioapic` variable depends on the `CONFIG_ACPI` and `CONFIG_X86_LOCAL_APIC` Linux kernel configuration options. If these options did not set, this variable will be just zero:

```
#define acpi_ioapic 0
```

The second condition - `!of_ioapic && nr_legacy_irqs()` checks that we do not use Open Firmware `I/O APIC` and legacy interrupt controller. We already know about the `nr_legacy_irqs`. The second is `of_ioapic` variable defined in the arch/x86/kernel/devicetree.c and initialized in the `dtb_ioapic_setup` function that build information about `APICs` in the devicetree. Note that `of_ioapic` variable depends on the `CONFIG_OF` Linux kernel configuration option. If this option is not set, the value of the `of_ioapic` will be zero too:

```
#ifdef CONFIG_OF
extern int of_ioapic;
...
...
...
#else
#define of_ioapic 0
...
...
...
#endif
```

If the condition will return non-zero value we call the:

```
setup_irq(2, &irq2);
```

function. First of all about the `irq2`. The `irq2` is the `irqaction` structure that defined in the arch/x86/kernel/irqinit.c source code file and represents `IRQ 2` line that is used to query devices connected cascade:

```
static struct irqaction irq2 = {
    .handler = no_action,
    .name = "cascade",
    .flags = IRQF_NO_THREAD,
};
```

Some time ago interrupt controller consisted of two chips and one was connected to second. The second chip that was connected to the first chip via this `IRQ 2` line. This chip serviced lines from `8` to `15` and after this lines of the first chip. So, for example Intel 8259A has following lines:

- `IRQ 0` - system time;
- `IRQ 1` - keyboard;
- `IRQ 2` - used for devices which are cascade connected;
- `IRQ 8` - RTC;
- `IRQ 9` - reserved;
- `IRQ 10` - reserved;
- `IRQ 11` - reserved;
- `IRQ 12` - `ps/2` mouse;
- `IRQ 13` - coprocessor;
- `IRQ 14` - hard drive controller;
- `IRQ 1` - reserved;
- `IRQ 3` - `COM2` and `COM4`;
- `IRQ 4` - `COM1` and `COM3`;
- `IRQ 5` - `LPT2`;
- `IRQ 6` - drive controller;
- `IRQ 7` - `LPT1`.

The `setup_irq` function defined in the kernel/irq/manage.c and takes two parameters:

- vector number of an interrupt;
- `irqaction` structure related with an interrupt.

This function initializes interrupt descriptor from the given vector number at the beginning:

```
struct irq_desc *desc = irq_to_desc(irq);
```

And call the `__setup_irq` function that setups given interrupt:

```
chip_bus_lock(desc);
retval = __setup_irq(irq, desc, act);
chip_bus_sync_unlock(desc);
return retval;
```

Note that the interrupt descriptor is locked during `__setup_irq` function will work. The `__setup_irq` function makes many different things: It creates a handler thread when a thread function is supplied and the interrupt does not nest into another interrupt thread, sets the flags of the chip, fills the `irqaction` structure and many many more.

All of the above it creates `/prov/vector_number` directory and fills it, but if you are using modern computer all values will be zero there:

```
$ cat /proc/irq/2/node
0

$cat /proc/irq/2/affinity_hint
00

cat /proc/irq/2/spurious
count 0
unhandled 0
last_unhandled 0 ms
```

because probably `APIC` handles interrupts on the our machine.

That's all.

Conclusion

It is the end of the eighth part of the Interrupts and Interrupt Handling chapter and we continued to dive into external hardware interrupts in this part. In the previous part we started to do it and saw early initialization of the `IRQs`. In this part we already saw non-early interrupts initialization in the `init_IRQ` function. We saw initialization of the `vector_irq` per-cpu array which is store vector numbers of the interrupts and will be used during interrupt handling and initialization of other stuff which is related to the external hardware interrupts.

In the next part we will continue to learn interrupts handling related stuff and will see initialization of the `softirqs`.

If you have any questions or suggestions write me a comment or ping me at twitter.

Please note that English is not my first language, And I am really sorry for any inconvenience. If you find any mistakes please send me PR to linux-insides.

Links

- IRQ
- percpu
- x86_64
- Intel 8259
- Programmable Interrupt Controller
- ISA
- MultiProcessor Configuration Table
- Local APIC
- I/O APIC
- SMP
- Inter-processor interrupt
- ternary operator
- gcc
- calling convention
- PDF. System V Application Binary Interface AMD64
- Call stack
- Open Firmware
- devicetree
- RTC
- Previous part

中断和中断处理(九)

延后中断(软中断，Tasklets 和工作队列)介绍

这是 Linux 内核中断和中断处理的第九节，在上一节我们分析了源文件 arch/x86/kernel/irqinit.c 中的 `init_IRQ` 实现。接下来的这一节我们将继续深入学习外部硬件中断的初始化。

中断处理会有一些特点，其中最主要的两个是：

- 中断处理必须快速执行完毕
- 有时中断处理必须做很多冗长的事情

就像你所想到的，我们几乎不可能同时做到这两点，之前的中断被分为两部分：

- 前半部
- 后半部

`后半部` 曾经是 Linux 内核延后中断执行的一种方式，但现在的实际情况已经不是这样了。现在它已作为一个遗留称谓代表内核中所有延后中断的机制。如你所知，中断处理代码运行于中断处理上下文中，此时禁止响应后续的中断，所以要避免中断处理代码长时间执行。但有些中断却又需要执行很多工作，所以中断处理有时会被分为两部分。第一部分中，中断处理先只做尽量少的重要工作，接下来提交第二部分给内核调度，然后就结束运行。当系统比较空闲并且处理器上下文允许处理中断时，第二部分被延后的剩余任务就会开始执行。

当前实现延后中断的有如下三种途径：

- `软中断`
- `tasklets`
- `工作队列`

在这一小节我们将详细介绍这三种实现，现在是时间深入了解一下了。

软中断

伴随着内核对并行处理的支持，出于性能考虑，所有新的下半部实现方案都基于被称之为 `ksoftirqd` (稍后将详细讨论)的内核线程。每个处理器都有自己的内核线程，名字叫做 `ksoftirqd/n` ，n是处理器的编号。我们可以通过系统命令 `systemd-cgls` 看到它们：

```
$ systemd-cgls -k | grep ksoft
├─   3 [ksoftirqd/0]
├─  13 [ksoftirqd/1]
├─  18 [ksoftirqd/2]
├─  23 [ksoftirqd/3]
├─  28 [ksoftirqd/4]
├─  33 [ksoftirqd/5]
├─  38 [ksoftirqd/6]
├─  43 [ksoftirqd/7]
```

由 `spawn_ksoftirqd` 函数启动这些线程。就像我们看到的，这个函数在早期的 initcall 被调用。

```
early_initcall(spawn_ksoftirqd);
```

软中断在 Linux 内核编译时就静态地确定了。`open_softirq` 函数负责 `softirq` 初始化，它在 kernel/softirq.c 中定义：

```
void open_softirq(int nr, void (*action)(struct softirq_action *))
{
    softirq_vec[nr].action = action;
}
```

这个函数有两个参数：

- `softirq_vec` 数组的索引序号
- 一个指向软中断处理函数的指针

我们首先来看 `softirq_vec` 数组：

```
static struct softirq_action softirq_vec[NR_SOFTIRQS] __cacheline_aligned_in_smp;
```

它在同一源文件中定义。`softirq_vec` 数组包含了 `NR_SOFTIRQS` (其值为10)个不同 `softirq` 类型的 `softirq_action`。当前版本的 Linux 内核定义了十种软中断向量。其中两个 tasklet 相关，两个网络相关，两个块处理相关，两个定时器相关，另外调度器和 RCU 也各占一个。所有这些都在一个枚举中定义：

```
enum
{
        HI_SOFTIRQ=0,
        TIMER_SOFTIRQ,
        NET_TX_SOFTIRQ,
        NET_RX_SOFTIRQ,
        BLOCK_SOFTIRQ,
        BLOCK_IOPOLL_SOFTIRQ,
        TASKLET_SOFTIRQ,
        SCHED_SOFTIRQ,
        HRTIMER_SOFTIRQ,
        RCU_SOFTIRQ,
        NR_SOFTIRQS
};
```

以上软中断的名字在如下的数组中定义：

```
const char * const softirq_to_name[NR_SOFTIRQS] = {
        "HI", "TIMER", "NET_TX", "NET_RX", "BLOCK", "BLOCK_IOPOLL",
        "TASKLET", "SCHED", "HRTIMER", "RCU"
};
```

我们也可以在 `/proc/softirqs` 的输出中看到他们：

```
~$ cat /proc/softirqs
                    CPU0       CPU1       CPU2       CPU3       CPU4       CPU5
CPU6       CPU7
          HI:          5          0          0          0          0          0
   0          0
       TIMER:     332519     310498     289555     272913     282535     279467     28
2895     270979
      NET_TX:       2320          0          0          2          1          1
   0          0
      NET_RX:     270221        225        338        281        311        262
 430        265
       BLOCK:     134282         32         40         10         12          7
   8          8
BLOCK_IOPOLL:          0          0          0          0          0          0
   0          0
     TASKLET:     196835          2          3          0          0          0
   0          0
       SCHED:     161852     146745     129539     126064     127998     128014     12
0243     117391
     HRTIMER:          0          0          0          0          0          0
   0          0
         RCU:     337707     289397     251874     239796     254377     254898     26
7497     256624
```

可以看到 `softirq_vec` 数组的类型为 `softirq_action` 。这是软中断机制里一个重要的数据结构，它只有一个指向中断处理函数的成员：

```
struct softirq_action
{
        void    (*action)(struct softirq_action *);
};
```

现在我们可以理解到 `open_softirq` 函数实际上用 `softirq_action` 参数填充了 `softirq_vec` 数组。由 `open_softirq` 注册的延后中断处理函数会由 `raise_softirq` 调用。这个函数只有一个参数 -- 软中断序号 `nr` 。来看下它的实现：

```
void raise_softirq(unsigned int nr)
{
        unsigned long flags;

        local_irq_save(flags);
        raise_softirq_irqoff(nr);
        local_irq_restore(flags);
}
```

可以看到在 `local_irq_save` 和 `local_irq_restore` 两个宏中间调用了 `raise_softirq_irqoff` 函数。 `local_irq_save` 的定义位于 include/linux/irqflags.h 头文件，它保存了 eflags 寄存器中的 IF 标志位并且禁用了当前处理器的中断。 `local_irq_restore` 宏定义于相同头文件中，它做了完全相反的事情：装回之前保存的中断标志位然后允许中断。这里之所以要禁用中断是因为将要运行的 `softirq` 中断处理运行于中断上下文中。

`raise_softirq_irqoff` 函数设置当前处理器上和nr参数对应的软中断标志位(`__softirq_pending`)。这是通过以下代码做到的：

```
__raise_softirq_irqoff(nr);
```

然后，通过 `in_interrupt` 函数获得 `irq_count` 值。我们在这一章的第一小节已经知道它是用来检测一个 cpu 是否处于中断环境。如果我们处于中断上下文中，我们就退出 `raise_softirq_irqoff` 函数，装回 `IF` 标志位并允许当前处理器的中断。如果不在中断上下文中，就会调用 `wakeup_softirqd` 函数：

```
if (!in_interrupt())
    wakeup_softirqd();
```

`wakeup_softirqd` 函数会激活当前处理器上的 `ksoftirqd` 内核线程：

```
static void wakeup_softirqd(void)
{
    struct task_struct *tsk = __this_cpu_read(ksoftirqd);

    if (tsk && tsk->state != TASK_RUNNING)
        wake_up_process(tsk);
}
```

每个 `ksoftirqd` 内核线程都运行 `run_ksoftirqd` 函数来检测是否有延后中断需要处理，如果有的话就会调用 `__do_softirq` 函数。`__do_softirq` 读取当前处理器对应的 `__softirq_pending` 软中断标记，并调用所有已被标记中断对应的处理函数。在执行一个延后函数的同时，可能会发生新的软中断。这会导致用户态代码由于 `__do_softirq` 要处理很多延后中断而很长时间不能返回。为了解决这个问题，系统限制了延后中断处理的最大耗时：

```
unsigned long end = jiffies + MAX_SOFTIRQ_TIME;
...
...
...
restart:
while ((softirq_bit = ffs(pending))) {
    ...
    h->action(h);
    ...
}
...
...
...
pending = local_softirq_pending();
if (pending) {
    if (time_before(jiffies, end) && !need_resched() &&
        --max_restart)
            goto restart;
}
...
```

除周期性检测是否有延后中断需要执行之外，系统还会在一些关键时间点上检测。一个主要的检测时间点就是当定义在 arch/x86/kernel/irq.c 的 `do_IRQ` 函数被调用时，这是 Linux 内核中执行延后中断的主要时机。在这个函数将要完成中断处理时它会调用 arch/x86/include/asm/apic.h 中定义的 `exiting_irq` 函数，`exiting_irq` 又调用了 `irq_exit` 。`irq_exit` 函数会检测当前处理器上下文是否有延后中断，有的话就会调用 `invoke_softirq` ：

```
if (!in_interrupt() && local_softirq_pending())
    invoke_softirq();
```

这样就调用到了我们上面提到的 `__do_softirq` 。每个 `softirq` 都有如下的阶段：通过 `open_softirq` 函数注册一个软中断，通过 `raise_softirq` 函数标记一个软中断来激活它，然后所有被标记的软中断将会在 Linux 内核下一次执行周期性软中断检测时得以调度，对应此类型软中断的处理函数也就得以执行。

从上述可看出，软中断是静态分配的，这对于后期加载的内核模块将是一个问题。基于软中断实现的 `tasklets` 解决了这个问题。

Tasklets

如果你阅读 Linux 内核源码中软中断相关的代码，你会发现它很少会被用到。内核中实现延后中断的主要途径是 `tasklets` 。正如上面说的， `tasklets` 构建于 `softirq` 中断之上，他是基于下面两个软中断实现的：

- `TASKLET_SOFTIRQ` ;
- `HI_SOFTIRQ` .

简而言之， `tasklets` 是运行时分配和初始化的软中断。和软中断不同的是，同一类型的 `tasklets` 可以在同一时间运行于不同的处理器上。我们已经了解到一些关于软中断的知识，当然上面的文字并不能详细讲解所有的细节，但我们现在可以通过直接阅读代码一步步的更深入了解软中断。我们返回到开始部分讨论的 `softirq_init` 函数实现，这个函数在 kernel/softirq.c 中定义如下：

```
void __init softirq_init(void)
{
        int cpu;

        for_each_possible_cpu(cpu) {
                per_cpu(tasklet_vec, cpu).tail =
                        &per_cpu(tasklet_vec, cpu).head;
                per_cpu(tasklet_hi_vec, cpu).tail =
                        &per_cpu(tasklet_hi_vec, cpu).head;
        }

        open_softirq(TASKLET_SOFTIRQ, tasklet_action);
        open_softirq(HI_SOFTIRQ, tasklet_hi_action);
}
```

可以看到在函数开头定义了一个名为 cpu 的 integer 类型变量。接下来他会作为参数传递给宏 `for_each_possible_cpu` 来获得系统中所有的处理器。如果 `possible_cpu` 对你来说是一个新的术语，你可以阅读 CPU masks 章节来了解更多知识。简单的说， `possible_cpu` 是系统运行期间插入的处理器集合。所有的 `possible processor` 存储在 `cpu_possible_bits` 位图中，你可以在 kernel/cpu.c 中找到他的定义：

```
static DECLARE_BITMAP(cpu_possible_bits, CONFIG_NR_CPUS) __read_mostly;
...
...
...
const struct cpumask *const cpu_possible_mask = to_cpumask(cpu_possible_bits);
```

好了，我们定义了 integer 类型变量 `cpu` 并且通过 `for_each_possible_cpu` 宏遍历了所有处理器，初始化了两个 `per-cpu` 变量：

- `tasklet_vec`；
- `tasklet_hi_vec`；

这两个 `per-cpu` 变量和 `softirq_init` 函数都定义在相同代码中，他们被定义为 `tasklet_head` 类型：

```
static DEFINE_PER_CPU(struct tasklet_head, tasklet_vec);
static DEFINE_PER_CPU(struct tasklet_head, tasklet_hi_vec);
```

`tasklet_head` 结构代表一组 `Tasklets`，它包含两个成员，head 和 tail：

```
struct tasklet_head {
        struct tasklet_struct *head;
        struct tasklet_struct **tail;
};
```

`tasklet_struct` 数据类型在 include/linux/interrupt.h 中定义，它代表一个 `Tasklet`。这本书之前部分我们没有见过这个单词，那我们先试着理解一下 `Tasklet` 究竟为何物。实际上，`Tasklet` 是处理延后中断的一种机制，来看一下 `tasklet_struct` 的具体定义：

```
struct tasklet_struct
{
        struct tasklet_struct *next;
        unsigned long state;
        atomic_t count;
        void (*func)(unsigned long);
        unsigned long data;
};
```

这个数据结构包含有下面5个成员：

- 调度队列中的下一个 `Tasklet`
- 当前这个 `Tasklet` 的状态
- 这个 `Tasklet` 是否处于活动状态
- `Tasklet` 的回调函数

- 回调函数的参数

上面代码中，在 `softirq_init` 函数中初始化了两个 tasklets 数组：`tasklet_vec` 和 `tasklet_hi_vec` 。Tasklets 和高优先级 Tasklets 分别存储于这两个数组中。初始化完成后我们看到代码 kernel/softirq.c 在 `softirq_init` 函数的最后又两次调用了 `open_softirq` ：

```
open_softirq(TASKLET_SOFTIRQ, tasklet_action);
open_softirq(HI_SOFTIRQ, tasklet_hi_action);
```

`open_softirq` 函数的主要作用是初始化软中断，接下来让我们看看它是怎么做的。和 Tasklets 相关的软中断处理函数有两个，分别是 `tasklet_action` 和 `tasklet_hi_action` 。其中 `tasklet_hi_action` 和 `HI_SOFTIRQ` 关联在一起，`tasklet_action` 和 `TASKLET_SOFTIRQ` 关联在一起。

Linux 内核提供一些 API 供操作 Tasklets 之用。首先是 `tasklet_init` 函数，它接受一个 `task_struct` 数据结构，一个处理函数，和另外一个参数，并利用这些参数来初始化所给的 `task_struct` 结构：

```
void tasklet_init(struct tasklet_struct *t,
                  void (*func)(unsigned long), unsigned long data)
{
    t->next = NULL;
    t->state = 0;
    atomic_set(&t->count, 0);
    t->func = func;
    t->data = data;
}
```

另外还有如下两个宏可以静态地初始化一个 tasklet：

```
DECLARE_TASKLET(name, func, data);
DECLARE_TASKLET_DISABLED(name, func, data);
```

Linux 内核提供三个函数标记一个 tasklet 已经准备就绪：

```
void tasklet_schedule(struct tasklet_struct *t);
void tasklet_hi_schedule(struct tasklet_struct *t);
void tasklet_hi_schedule_first(struct tasklet_struct *t);
```

第一个函数使用普通优先级调度一个 tasklet，第二个使用高优先级，第三个则用更高优先级。所有这三个函数的实现都很类似，所以我们只看一下第一个 `tasklet_schedule` 的实现：

```
static inline void tasklet_schedule(struct tasklet_struct *t)
{
    if (!test_and_set_bit(TASKLET_STATE_SCHED, &t->state))
        __tasklet_schedule(t);
}

void __tasklet_schedule(struct tasklet_struct *t)
{
        unsigned long flags;

        local_irq_save(flags);
        t->next = NULL;
        *__this_cpu_read(tasklet_vec.tail) = t;
        __this_cpu_write(tasklet_vec.tail, &(t->next));
        raise_softirq_irqoff(TASKLET_SOFTIRQ);
        local_irq_restore(flags);
}
```

我们看到它检测并设置所给的 tasklet 为 `TASKLET_STATE_SCHED` 状态，然后以所给 tasklet 为参数执行了 `__tasklet_schedule` 函数。 `__tasklet_schedule` 看起来和前面见到的 `raise_softirq` 很像。一开始它保存中断标志并禁用中断，继而将新的 tasklet 添加到 `tasklet_vec`，然后调用了我们前面见过的 `raise_softirq_irqoff` 函数。当 Linux 内核调度器决定去运行一个延后函数， `tasklet_action` 函数会被作为和 `TASKLET_SOFTIRQ` 相关联的延后函数调用。同样的， `tasklet_hi_action` 会被作为和 `HI_SOFTIRQ` 相关联的延后函数调用。这些函数之所以如此相似是因为他们之间只有一个地方不同 --- `tasklet_action` 使用 `tasklet_vec` 而 `tasklet_hi_action` 使用 `tasklet_hi_vec`。

让我们看下 `tasklet_action` 函数的实现：

```
static void tasklet_action(struct softirq_action *a)
{
    local_irq_disable();
    list = __this_cpu_read(tasklet_vec.head);
    __this_cpu_write(tasklet_vec.head, NULL);
    __this_cpu_write(tasklet_vec.tail, this_cpu_ptr(&tasklet_vec.head));
    local_irq_enable();

    while (list) {
        if (tasklet_trylock(t)) {
            t->func(t->data);
            tasklet_unlock(t);
        }
        ...
        ...
        ...
    }
}
```

在 `tasklet_action` 开始时利用 `local_irq_disable` 宏禁用了当前处理器的中断(你可以阅读本书第二部分了解更多关于此宏的信息)。接下来获取到当前处理器对应的普通优先级 tasklet 列表并把它设置为 `NULL` ，这是因为所有的 tasklet 都将被执行。然后使能当前处理器的中断，循环遍历 tasklet 列表，每一次遍历都会对当前 tasklet 调用 `tasklet_trylock` 函数来更新它的状态为 `TASKLET_STATE_RUN` ：

```
static inline int tasklet_trylock(struct tasklet_struct *t)
{
    return !test_and_set_bit(TASKLET_STATE_RUN, &(t)->state);
}
```

如果这个操作成功了就会执行此 tasklet 的处理函数(我们在 `tasklet_init` 中所设置的)，然后调用 `tasklet_unlock` 函数清除他的 `TASKLET_STATE_RUN` 状态。

通常情况下，这就是 `tasklet` 的所有概念。当然这些还不足以覆盖所有的 `tasklets` ，但是我想大家可以以此为切入点继续学习下去。

`tasklets` 在 Linux 内核中是一个广泛使用的概念，但就像我在本章开头所写的，还有第三个延后中断机制 -- 工作队列。接下来我们将会看看它又是怎样一种机制。

工作队列

`工作队列` 是另外一个处理延后函数的概念，它大体上和 `tasklets` 类似。工作队列运行于内核进程上下文，而 `tasklets` 运行于软中断上下文。这意味着 `工作队列` 函数不必像 `tasklets` 一样必须是原子性的。Tasklets 总是运行于它提交自的那个处理器，工作队列在默认情况下也是这样。`工作队列` 在 Linux 内核代码 kernel/workqueue.c 中由如下的数据结构表示：

```
struct worker_pool {
    spinlock_t              lock;
    int                     cpu;
    int                     node;
    int                     id;
    unsigned int            flags;

    struct list_head        worklist;
    int                     nr_workers;
...
...
...
```

因为这个结构有非常多的成员，这里就不把它们全部罗列出来，下面只讨论上面列出的这几个。

工作队列最基础的用法，是作为创建内核线程的接口来处理提交到队列里的工作任务。所有这些内核线程称之为 `worker thread` 。工作队列内的任务是由代码 include/linux/workqueue.h 中定义的 `work_struct` 表示的，起定义如下：

```
struct work_struct {
    atomic_long_t data;
    struct list_head entry;
    work_func_t func;
#ifdef CONFIG_LOCKDEP
    struct lockdep_map lockdep_map;
#endif
};
```

这里有两个字段比较有意思： `func` --将被 `工作队列` 调度执行的函数， `data` --这个函数的参数。Linux 内核提供了称之为 `kworker` 的特定于每个 cpu 的内核线程：

```
systemd-cgls -k | grep kworker
├    5 [kworker/0:0H]
├   15 [kworker/1:0H]
├   20 [kworker/2:0H]
├   25 [kworker/3:0H]
├   30 [kworker/4:0H]
...
...
...
```

这些线程会被用来调度执行工作队列的延后函数(就像 `ksoftirqd` 之于 `软中断`)。除此之外我们还可以为一个 `工作队列` 创建一个新的工作线程。Linux 内核提供了如下宏静态创建一个队列任务：

```
#define DECLARE_WORK(n, f) \
    struct work_struct n = __WORK_INITIALIZER(n, f)
```

它需要两个参数：工作队列的名字和工作队列的函数。我们还可以在运行时动态创建：

```
#define INIT_WORK(_work, _func)       \
    __INIT_WORK((_work), (_func), 0)

#define __INIT_WORK(_work, _func, _onstack)                     \
    do {                                                        \
            __init_work((_work), _onstack);                     \
            (_work)->data = (atomic_long_t) WORK_DATA_INIT();   \
            INIT_LIST_HEAD(&(_work)->entry);                    \
             (_work)->func = (_func);                           \
    } while (0)
```

这个宏需要一个 `work_struct` 数据结构作为将要创建的队列任务，和一个将在这个任务里调度运行的函数。通过这两个宏的其中一个创建一个 `work` 后，我们需要把它放到 `工作队列` 中去。可以通过 `queue_work` 或者 `queue_delayed_work` 来做到这一点：

```
static inline bool queue_work(struct workqueue_struct *wq,
                              struct work_struct *work)
{
    return queue_work_on(WORK_CPU_UNBOUND, wq, work);
}
```

`queue_work` 只是调用了 `queue_work_on` 函数指定相应的处理器。注意这里给 `queue_work_on` 函数传递了 `WORK_CPU_UNBOUND` 参数，它作为代表队列任务要绑定到哪一个处理器的枚举一员，定义于 include/linux/workqueue.h。`queue_work_on` 函数测试并设置所给 `任务` 的 `WORK_STRUCT_PENDING_BIT` 标志位，然后以所给的工作队列和队列任务为参数执行 `__queue_work` 函数：

```
bool queue_work_on(int cpu, struct workqueue_struct *wq,
           struct work_struct *work)
{
    bool ret = false;
    ...
    if (!test_and_set_bit(WORK_STRUCT_PENDING_BIT, work_data_bits(work))) {
        __queue_work(cpu, wq, work);
        ret = true;
    }
    ...
    return ret;
}
```

`__queue_work` 函数得到参数 `work poll`。是的，是 `work poll` 而不是 `workqueue`。实际上，所有的 `works` 都没有放在 `workqueue` 中，而是放在 Linux 内核中由 `worker_pool` 数据结构所定义的 `work poll`。如上所述，`workqueue_struct` 数据结构的 `pwqs` 成员是一个 `worker_pool` 列表。当我们创建一个 `workqueue`，他针对每一个处理器都创建了 `worker_pool`。每一个和 `worker_pool` 相关联的 `pool_workqueue` 都分配在相同的处理器上对应的优先级队列，`workqueue` 通过他们和 `worker_pool` 交互。在 `__queue_work` 函数里使用 `raw_smp_processor_id` 设置 cpu 为当前处理器在第四章你可以找到更多相关信息)，得到与所给 `work_struct` 对应的 `pool_workqueue` 并将 `work` 插入到 `workqueue`：

```
static void __queue_work(int cpu, struct workqueue_struct *wq,
                         struct work_struct *work)
{
...
...
...
if (req_cpu == WORK_CPU_UNBOUND)
    cpu = raw_smp_processor_id();

if (!(wq->flags & WQ_UNBOUND))
    pwq = per_cpu_ptr(wq->cpu_pwqs, cpu);
else
    pwq = unbound_pwq_by_node(wq, cpu_to_node(cpu));
...
...
...
insert_work(pwq, work, worklist, work_flags);
```

现在我们可以创建 `works` 和 `workqueue` ，我们需要知道他们究竟会在何时被执行。就像前面提到的，所有的 `works` 都会在内核线程中执行。当内核线程得到调度，它开始执行 `workqueue` 中的 `works` 。每一个工作队列内核线程都会在 `worker_thread` 函数里执行一个循环。这些内核线程会做很多不同的事情，其中一些和本章前面提到的很类似。当开始执行时，所有的 `work_struct` 和 `works` 都会从他的 `workqueue` 移除。

总结

现在结束了中断和中断处理的第九节。这一节中我们继续讨论了外部硬件中断。在之前部分我们看到了 `IRQs` 的初始化和 `irq_desc` 数据结构，在这一节我们看到了用于延后函数的三个概念： `软中断` ， `tasklet` 和 `工作队列` 。

下一节将是 `中断和中断处理` 的最后一节。我们将会了解真正的硬件驱动，并试着学习它是怎样和中断子系统一起工作的。

如果你有任何问题或建议，请给我发评论或者给我发 Twitter。

请注意英语并不是我的母语，我为任何表达不清楚的地方感到抱歉。如果你发现任何错误请发 **PR** 到 **linux-insides**。**(**译者注：翻译问题请发 **PR** 到 **linux-insides-cn)**

链接

- initcall
- IF
- eflags

* CPU masks
* per-cpu
* Workqueue
* Previous part

中断和中断处理(十)

终结篇

本文是 Linux 内核中断和中断处理的第十节。在上一节，我们了解了延后中断及其相关概念，如 `softirq` ，`tasklet` ，`workqueue` 。本节我们继续深入这个主题，现在是见识真正的硬件驱动的时候了。

以 StringARM** SA-100/21285 评估板串行驱动为例，我们来观察驱动程序如何请求一个 IRQ 线，一个中断被触发时会发生什么之类的。驱动程序代码位于 drivers/tty/serial/21285.c 源文件。好啦，源码在手，说走就走！

一个内核模块的初始化

与本书其他新概念类似，为了考察这个驱动程序，我们从考察它的初始化过程开始。如你所知，Linux 内核为驱动程序或者内核模块的初始化和终止提供了两个宏：

- `module_init`
- `module_exit`

可以在驱动程序的源代码中查阅这些宏的用法:

```
module_init(serial21285_init);
module_exit(serial21285_exit);
```

大多数驱动程序都能编译成一个可装载的内核模块，亦或被静态地链入 Linux 内核。前一种情况下，一个设备驱动程序的初始化由 `module_init` 与 `module_exit` 宏触发。这些宏定义在 include/linux/init.h 中:

```
#define module_init(initfn)                                     \
        static inline initcall_t __inittest(void)               \
        { return initfn; }                                      \
        int init_module(void) __attribute__((alias(#initfn)));

#define module_exit(exitfn)                                     \
        static inline exitcall_t __exittest(void)               \
        { return exitfn; }                                      \
        void cleanup_module(void) __attribute__((alias(#exitfn)));
```

并被 initcall 函数调用：

- `early_initcall`
- `pure_initcall`
- `core_initcall`
- `postcore_initcall`
- `arch_initcall`
- `subsys_initcall`
- `fs_initcall`
- `rootfs_initcall`
- `device_initcall`
- `late_initcall`

这些函数又被 init/main.c 中的 `do_initcalls` 函数调用。然而，如果设备驱动程序被静态链入 Linux 内核，那么这些宏的实现则如下所示：

```
#define module_init(x)  __initcall(x);
#define module_exit(x)  __exitcall(x);
```

这种情况下，模块装载的实现位于 kernel/module.c 源文件中，而初始化发生在 `do_init_module` 函数内。我们不打算在本章深入探讨可装载模块的细枝末节，而会在一个专门介绍 Linux 内核模块的章节中窥其真容。话说回来，`module_init` 宏接受一个参数 - 本例中这个值是 `serial21285_init` 。从函数名可以得知，这个函数做了一些驱动程序初始化的相关工作。请看：

```
static int __init serial21285_init(void)
{
    int ret;

    printk(KERN_INFO "Serial: 21285 driver\n");

    serial21285_setup_ports();

    ret = uart_register_driver(&serial21285_reg);
    if (ret == 0)
        uart_add_one_port(&serial21285_reg, &serial21285_port);

    return ret;
}
```

如你所见，首先它把驱动程序相关信息写入内核缓冲区，然后调用 `serial21285_setup_ports` 函数。该函数设置了 `serial21285_port` 设备的基本 uart 时钟：

```
unsigned int mem_fclk_21285 = 50000000;

static void serial21285_setup_ports(void)
{
    serial21285_port.uartclk = mem_fclk_21285 / 4;
}
```

此处的 `serial21285` 是描述 `uart` 驱动程序的结构体：

```
static struct uart_driver serial21285_reg = {
    .owner            = THIS_MODULE,
    .driver_name    = "ttyFB",
    .dev_name        = "ttyFB",
    .major            = SERIAL_21285_MAJOR,
    .minor            = SERIAL_21285_MINOR,
    .nr                = 1,
    .cons            = SERIAL_21285_CONSOLE,
};
```

如果驱动程序注册成功，我们借助 drivers/tty/serial/serial_core.c 源文件中的 `uart_add_one_port` 函数添加由驱动程序定义的端口 `serial21285_port` 结构体，然后从 `serial21285_init` 函数返回：

```
if (ret == 0)
    uart_add_one_port(&serial21285_reg, &serial21285_port);

return ret;
```

到此为止，我们的驱动程序初始化完毕。当一个 `uart` 端口被 drivers/tty/serial/serial_core.c 中的 `uart_open` 函数打开，该函数会调用 `uart_startup` 函数来启动这个串行端口，后者会调用 `startup` 函数。它是 `uart_ops` 结构体的一部分。每个 `uart` 驱动程序都会定义这样一个结构体。在本例中，它是这样的：

```
static struct uart_ops serial21285_ops = {
    ...
    .startup    = serial21285_startup,
    ...
}
```

可以看到， `.startup` 字段是对 `serial21285_startup` 函数的引用。这个函数的实现是我们的关注重点，因为它与中断和中断处理密切相关。

请求中断线

我们来看看 `serial21285_startup` 函数的实现：

```
static int serial21285_startup(struct uart_port *port)
{
    int ret;

    tx_enabled(port) = 1;
    rx_enabled(port) = 1;

    ret = request_irq(IRQ_CONRX, serial21285_rx_chars, 0,
              serial21285_name, port);
    if (ret == 0) {
        ret = request_irq(IRQ_CONTX, serial21285_tx_chars, 0,
                  serial21285_name, port);
        if (ret)
            free_irq(IRQ_CONRX, port);
    }

    return ret;
}
```

首先是 `TX` 和 `RX` 。一个设备的串行总线仅由两条线组成：一条用于发送数据，另一条用于接收数据。与此对应，串行设备应该有两个串行引脚：接收器 - `RX` 和发送器 - `TX` 。通过调用 `tx_enabled` 和 `rx_enalbed` 这两个宏来激活这些线。函数接下来的部分是我们最感兴趣的。注意 `request_irq` 这个函数。它注册了一个中断处理程序，然后激活一条给定的中断线。看一下这个函数的实现细节。该函数定义在 include/linux/interrupt.h 头文件中，如下所示：

```
static inline int __must_check
request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
            const char *name, void *dev)
{
        return request_threaded_irq(irq, handler, NULL, flags, name, dev);
}
```

可以看到， `request_irq` 函数接受五个参数：

- `irq` - 被请求的中断号
- `handler` - 中断处理程序指针
- `flags` - 掩码选项
- `name` - 中断拥有者的名称
- `dev` - 用于共享中断线的指针

现在我们来考察 `request_irq` 函数的调用。可以看到，第一个参数是 `IRQ_CONRX` 。我们知道它是中断号，但 `CONRX` 又是什么东西？这个宏定义在 arch/arm/mach-footbridge/include/mach/irqs.h 头文件中。我们可以在这里找到 `21285` 主板能够产生的全部

中断。注意，在第二次调用 `request_irq` 函数时，我们传入了 `IRQ_CONTX` 中断号。我们的驱动程序会在这些中断中处理 `RX` 和 `TX` 事件。这些宏的实现很简单：

```
#define IRQ_CONRX               _DC21285_IRQ(0)
#define IRQ_CONTX               _DC21285_IRQ(1)
...
...
...
#define _DC21285_IRQ(x)         (16 + (x))
```

这个主板的 ISA 中断号分布在 `0` 到 `15` 这个范围内。因此，我们的中断号就是在此之后的头两个值： `16` 和 `17` 。在 `request_irq` 函数的两次调用中，第二个参数分别是 `serial21285_rx_chars` 和 `serial21285_tx_chars` 函数。当一个 `RX` 或 `TX` 中断发生时，这些函数就会被调用。我们不会在此深入探究这些函数，因为本章讲述的是中断与中断处理，而并非设备和驱动。下一个参数是 `flags` ， `request_irq` 函数的两次调用中，它的值都是零。所有合法的 `flags` 都在 include/linux/interrupt.h 中定义成诸如 `IRQF_*` 此类的宏。一些例子：

- `IRQF_SHARED` - 允许多个设备共享此中断号
- `IRQF_PERCPU` - 此中断号属于单独cpu的(per cpu)
- `IRQF_NO_THREAD` - 中断不能线程化
- `IRQF_NOBALANCING` - 此中断步参与irq平衡时
- `IRQF_IRQPOLL` - 此中断用于轮询
- 等等

这里，我们传入的是 `0` ，也就是 `IRQF_TRIGGER_NONE` 。这个标志是说，它不配置任何水平触发或边缘触发的中断行为。至于第四个参数(`name`)，我们传入 `serial21285_name` ，它定义如下：

```C
static const char serial21285_name[] = "Footbridge UART";
```

它会显示在 `/proc/interrupts` 的输出中。针对最后一个参数，我们传入一个指向 `uart_port` 结构体的指针。对 `request_irq` 函数及其参数有所了解后，我们来看看它的实现。从上文可知， `request_irq` 函数内部只是调用了定义在 kernel/irq/manage.c 源文件中的 `request_threaded_irq` 函数，并分配了一个给定的中断线。该函数起始部分是 `irqaction` 和 `irq_desc` 的定义：

```
int request_threaded_irq(unsigned int irq, irq_handler_t handler,
                         irq_handler_t thread_fn, unsigned long irqflags,
                         const char *devname, void *dev_id)
{
        struct irqaction *action;
        struct irq_desc *desc;
        int retval;
        ...
        ...
        ...
}
```

在本章，我们已经见识过 `irqaction` 和 `irq_desc` 结构体了。第一个结构体表示一个中断动作描述符，它包含中断处理程序指针，设备名称，中断号等等。第二个结构体表示一个中断描述符，包含指向 `irqaction` 的指针，中断标志等等。注意，`request_threaded_irq` 函数被 `request_irq` 调用时，带了一个额外的参数：`irq_handler_t thread_fn` 。如果这个参数不为 `NULL` ，它会创建 `irq` 线程，并在该线程中执行给定的 `irq` 处理程序。下一步，我们要做如下检查：

```
if (((irqflags & IRQF_SHARED) && !dev_id) ||
            (!(irqflags & IRQF_SHARED) && (irqflags & IRQF_COND_SUSPEND)) ||
            ((irqflags & IRQF_NO_SUSPEND) && (irqflags & IRQF_COND_SUSPEND)))
               return -EINVAL;
```

首先，我们确保共享中断时传入了真正的 `dev_id` (译者注：不然后面搞不清楚哪台设备产生了中断)，而且 `IRQF_COND_SUSPEND` 仅对共享中断生效。否则退出函数，返回 `-EINVAL` 错误。之后，我们借助 kernel/irq/irqdesc.c 源文件中定义的 `irq_to_desc` 函数将给定的 `irq` 中断号转换成 `irq` 中断描述符。如果不成功，则退出函数，返回 `-EINVAL` 错误：

```
desc = irq_to_desc(irq);
if (!desc)
    return -EINVAL;
```

`irq_to_desc` 函数检查给定的 `irq` 中断号是否小于最大中断号，并且返回中断描述符。这里，`irq` 中断号就是 `irq_desc` 数组的偏移量：

```
struct irq_desc *irq_to_desc(unsigned int irq)
{
        return (irq < NR_IRQS) ? irq_desc + irq : NULL;
}
```

由于我们已经把 `irq` 中断号转换成了 `irq` 中断描述符，现在来检查描述符的状态，确保我们可以请求中断：

```
if (!irq_settings_can_request(desc) || WARN_ON(irq_settings_is_per_cpu_devid(desc)))
    return -EINVAL;
```

失败则返回 `-EINVAL` 错误。接着，我们检查给定的中断处理程序(译者注：是指 `handler` 变量)。如果它没被传入 `request_irq` 函数，我们就检查 `thread_fn` 。两个都是 `NULL` 则返回 `-EINVAL` 。如果中断处理程序没有被传入 `request_irq` 函数而 `thread_fn` 不为空，则把 `handler` 设为 `irq_default_primary_handler` ：

```
if (!handler) {
    if (!thread_fn)
        return -EINVAL;
    handler = irq_default_primary_handler;
}
```

下一步，我们通过 `kzalloc` 函数为 `irqaction` 分配内存，若不成功则返回：

```
action = kzalloc(sizeof(struct irqaction), GFP_KERNEL);
if (!action)
    return -ENOMEM;
```

欲知 `kzalloc` 详情，请查阅专门介绍 Linux 内核内存管理的章节。为 `irqaction` 分配空间后，我们即对这个结构体进行初始化，设置它的中断处理程序，中断标志，设备名称等等：

```
action->handler = handler;
action->thread_fn = thread_fn;
action->flags = irqflags;
action->name = devname;
action->dev_id = dev_id;
```

在 `request_threaded_irq` 函数末尾，我们调用 kernel/irq/manage.c 中的 `__setup_irq` 函数，并注册一个给定的 `irqaction` 。然后释放 `irqaction` 内存并返回：

```
chip_bus_lock(desc);
retval = __setup_irq(irq, desc, action);
chip_bus_sync_unlock(desc);

if (retval)
    kfree(action);

return retval;
```

注意， `__setup_irq` 函数的调用位于 `chip_bus_lock` 和 `chip_bus_sync_unlock` 函数之间。这些函数对慢速总线(如 i2c)芯片进行锁定／解锁。现在来看看 `__setup_irq` 函数的实现。 `__setup_irq` 函数开头是各种检查。首先我们检查给定的中断描述符不为 `NULL` ， `irqchip` 不为 `NULL` ，以及给定的中断描述符模块拥有者不为 `NULL` 。接下来我们检查中断是否嵌套在其他中断线程中。如果是的，我们则以 `irq_nested_primary_handler` 替换 `irq_default_priamry_handler` 。

下一步，如果给定的中断不是嵌套的，并且 `thread_fn` 不为空，我们就通过 `kthread_create` 创建了一个中断处理线程。

```
if (new->thread_fn && !nested) {
    struct task_struct *t;
    t = kthread_create(irq_thread, new, "irq/%d-%s", irq, new->name);
    ...
}
```

并在最后为给定的中断描述符的剩余字段赋值。于是，我们的 `16` 和 `17` 号中断请求线注册完毕。当一个中断控制器获得这些中断的相关事件时， `serial21285_rx_chars` 和 `serial21285_tx_chars` 函数会被调用。现在我们来看一看一个中断发生时到底发生了什么。

准备处理中断

通过上文，我们观察了为给定的中断描述符请求中断号，为给定的中断注册 `irqaction` 结构体的过程。我们已经知道，当一个中断事件发生时，中断控制器向处理器通知该事件，处理器尝试为这个中断找到一个合适的中断门。如果你已阅读本章第八节，你应该还记得 `native_init_IRQ` 函数。这个函数会初始化本地 APIC。这个函数的如下部分是我们现在最感兴趣的地方：

```
for_each_clear_bit_from(i, used_vectors, first_system_vector) {
    set_intr_gate(i, irq_entries_start +
        8 * (i - FIRST_EXTERNAL_VECTOR));
}
```

这里，我们从第 `first_system_vector` 位开始，依次向后迭代 `used_vectors` 位图中所有被清除的位：

```
int first_system_vector = FIRST_SYSTEM_VECTOR; // 0xef
```

并且设置中断门，`i` 是向量号，`irq_entries_start + 8 * (i - FIRST_EXTERNAL_VECTOR)` 是起始地址。仅有一处尚不明了 - `irq_entries_start`。这个符号定义在 arch/x86/entry/entry_64.S 汇编文件中，并提供了 `irq` 入口。一起来看：

```
    .align 8
ENTRY(irq_entries_start)
    vector=FIRST_EXTERNAL_VECTOR
    .rept (FIRST_SYSTEM_VECTOR - FIRST_EXTERNAL_VECTOR)
    pushq    $(~vector+0x80)
    vector=vector+1
    jmp    common_interrupt
    .align    8
    .endr
END(irq_entries_start)
```

这里我们可以看到 GNU 汇编器的 `.rept` 指令。这条指令会把 `.endr` 之前的这几行代码重复 `FIRST_SYSTEM_VECTOR - FIRST_EXTERNAL_VECTOR` 次。我们已经知道 `FIRST_SYSTEM_VECTOR` 的值是 `0xef`，而 `FIRST_EXTERNAL_VECTOR` 等于 `0x20`。于是，它将运行：

```
>>> 0xef - 0x20
207
```

次。在 `.rept` 指令主体中，我们把入口程序地址压入栈中(注意，我们使用负数表示中断向量号，因为正数留作标识系统调用之用)，将 `vector` 变量加 1，并跳转到 `common_interrupt` 标签。在 `common_interrupt` 中，我们调整了栈中向量号，执行 `interrupt` 指令，参数是 `do_IRQ`：

```
common_interrupt:
    addq    $-0x80, (%rsp)
    interrupt do_IRQ
```

`interrupt` 宏定义在同一个源文件中。它把通用寄存器的值保存在栈中。如果需要，它还会通过 `SWAPGS` 汇编指令在内核中改变用户空间 `gs` 寄存器。它会增加 per-cpu 的 `irq_count` 变量，来表明我们处于中断状态，然后调用 `do_IRQ` 函数。该函数定义于 arch/x86/kernel/irq.c 源文件中，作用是处理我们的设备中断。让我们一起考察这个函数。`do_IRQ` 函数接受一个参数 - `pt_regs` 结构体，它存放着用户空间寄存器的值：

```
__visible unsigned int __irq_entry do_IRQ(struct pt_regs *regs)
{
    struct pt_regs *old_regs = set_irq_regs(regs);
    unsigned vector = ~regs->orig_ax;
    unsigned irq;

    irq_enter();
    exit_idle();
    ...
    ...
    ...
}
```

函数开头调用了 `set_irq_regs` 函数，后者返回被保存的 `per-cpu` 中断寄存器指针。然后又调用 `irq_enter` 和 `exit_idle` 函数。第一个函数 `irq_enter` 进入到一个中断上下文，更新 `__preempt_count` 变量。第二个函数 `exit_idle` 检查当前进程是否是 pid 为 `0` 的 `idle` 进程，然后把 `IDLE_END` 传送给 `idle_notifier` 。

接下来，我们从当前 cpu 中读取 `irq` 值，并调用 `handle_irq` 函数：

```
irq = __this_cpu_read(vector_irq[vector]);

if (!handle_irq(irq, regs)) {
    ...
    ...
    ...
}
...
...
...
```

`handle_irq` 函数定义于 arch/x86/kernel/irq_64.c 源文件中，它检查给定的中断描述符，然后调用 `generic_handle_irq_desc` 函数：

```
desc = irq_to_desc(irq);
    if (unlikely(!desc))
        return false;
generic_handle_irq_desc(irq, desc);
```

该函数又调用中断处理程序：

```
static inline void generic_handle_irq_desc(unsigned int irq, struct irq_desc *desc)
{
       desc->handle_irq(irq, desc);
}
```

但是，停一停...... `handle_irq` 是何方神圣，为什么在知道 `irqaction` 指向真正的中断处理程序的情况下，偏偏通过中断描述符调用我们的中断处理程序？实际上，`irq_desc->handle_irq` 是一个用来调用中断处理程序的上层 API。它在设备树 和 APIC 的初始化过程中就设定好了。内核通过它选择正确的函数以及 `irq->actions(s)` 的调用链。就这样，当一个中断发生时，`serial21285_tx_chars` 或者 `serial21285_rx_chars` 函数会被调用。

在 `do_IRQ` 函数末尾，我们调用 `irq_exit` 函数来退出中断上下文，调用 `set_irq_regs` 函数并传入先前的用户空间寄存器，最后返回：

```
irq_exit();
set_irq_regs(old_regs);
return 1;
```

我们已经知道，当一个 `IRQ` 工作结束之后，如果有延后中断，它们会被执行。

退出中断

好了，中断处理程序执行完毕，我们必须从中断中返回。在 `do_IRQ` 函数将工作处理完毕后，我们将回到 arch/x86/entry/entry_64.S 汇编代码的 `ret_from_intr` 标签处。首先，我们通过 `DISABLE_INTERRUPTS` 宏禁止中断，这个宏被扩展成 `cli` 指令，将 per-cpu 的 `irq_count` 变量值减 1。记住，当我们处于中断上下文的时候，这个变量的值是 `1`：

```
DISABLE_INTERRUPTS(CLBR_NONE)
TRACE_IRQS_OFF
decl    PER_CPU_VAR(irq_count)
```

最后一步，我们检查之前的上下文(用户空间或者内核空间)，正确地恢复它，然后通过指令退出中断：

```
INTERRUPT_RETURN
```

此处的 `INTERRUPT_RETURN` 宏是：

```
#define INTERRUPT_RETURN    jmp native_iret
```

而

```
ENTRY(native_iret)

.global native_irq_return_iret
native_irq_return_iret:
    iretq
```

本节到此结束。

总结

这里是中断和中断处理 章节的第十节的结尾。如你在本节开头读到的那样，这是本章的最后一节。本章开篇阐述了中断理论，我们于是明白了什么是中断，中断的类型，然后也了解了异常以及对这种类型中断的处理，延后中断。最后在本节，我们考察了硬件中断和对这些中断的处理。当然，本节甚至本章都未能覆盖到 Linux 内核中断和中断处理的所有方面。这样并不现实，至少对我而言如此。这是一项浩大工程，不知你作何感想，对我来说，它确实浩大。这个主题远远超出本章讲述的内容，我不确定地球上能否找到一本书可以涵盖这个主题。我们漏掉了关于中断和中断处理的很多内容，但我相信，深入研究中断和中断处理相关的内核源码是个不错的点子。

如果有任何疑问或者建议，撰写评论或者在 twitter 上联系我。

请注意，英语并非我的母语。任何不便之处，我深感抱歉。如果发现任何错误，请在 **linux-insides** 向我发送 **PR**。**(**译者注：翻译问题请发送 **PR** 到 **linux-insides-cn)**

链接

- 串行驱动文档
- StrongARM** SA-110/21285 评估板
- IRQ
- 模块
- initcall
- uart
- ISA
- 内存管理
- i2c
- APIC
- GNU 汇编器
- 处理器寄存器
- per-cpu
- pid

- 设备树
- 系统调用
- 上一节

系统调用

本章描述 Linux 内核中的系统调用概念。

- 系统调用概念简介 - 介绍 Linux 内核中的系统调用概念
- Linux 内核如何处理系统调用 - 介绍 Linux 内核如何处理来自于用户空间应用的系统调用。
- vsyscall and vDSO - 介绍 `vsyscall` 和 `vDSO` 概念。
- Linux 内核如何运行程序 - 介绍一个程序的启动过程。
- open 系统调用的实现 - 介绍 open 系统调用的实现。
- Linux 资源限制 - 介绍 `getrlimit/setrlimit` 的实现。

Linux 内核系统调用 第一节

简介

这次提交为 linux-insides 添加一个新的章节，从标题就可以知道, 这一章节将介绍Linux 内核中 System Call 的概念。章节内容的选择并非偶然。在前一章节我们了解了中断及中断处理。系统调用的概念与中断非常相似，这是因为软件中断是执行系统调用最常见的方式。我们将讨论系统调用概念的各个方面。例如，用户空间发起系统调用的细节，内核中一组系统调用处理器的执行过程, VDSO 和 vsyscall 概念以及其他信息。

在了解 Linux 内核系统调用执行过程之前，了解一些系统调用的原理是有帮助的。我们从下面的段落开始。

什么是系统调用?

系统调用是用户空间请求内核服务。操作系统内核提供很多服务。当程序读写文件，开始监听连接的socket ，删除或创建目录或程序结束时，都会执行系统调用。换句话说，系统调用仅仅是一些 [C] (https://en.wikipedia.org/wiki/C_%28programming_language%29) 内核空间函数，用户空间程序调用其处理一些请求。

Linux 内核提供一系列的函数并且这些函数与CPU架构相关。 例如：x86_64 提供 322 个系统调用，x86 提供 358 个不同的系统调用。 系统调用仅仅是一些函数。 我们讨论一个使用汇编语言编写的简单 `Hello world` 示例:

```
.data

msg:
    .ascii "Hello, world!\n"
    len = . - msg

.text
    .global _start

_start:
    movq  $1, %rax
    movq  $1, %rdi
    movq  $msg, %rsi
    movq  $len, %rdx
    syscall

    movq  $60, %rax
    xorq  %rdi, %rdi
    syscall
```

使用下面的命令可编译这些语句:

```
$ gcc -c test.S
$ ld -o test test.o
```

执行:

```
./test
Hello, world!
```

这些简单的代码是一个简单的Linux `x86_64` 架构 `Hello world` 汇编程序，代码包含两个段:

- `.data`
- `.text`

第一个段 - `.data` 存储程序的初始数据 (在示例中为`Hello world` 字符串). 第二个段 - `.text` 包含程序的代码. 程序可分为两部分: 第一部分为第一个 `syscall` 指令之前的代码，第二部分为两个 `syscall` 指令之间的代码。首先在示例程序及一般应用中， `syscall` 指令有什么功能?64-ia-32-architectures-software-developer-vol-2b-manual中提到:

```
SYSCALL 引起操作系统系统调用处理器处于特权级0，通过加载IA32_LSTAR MSR至RIP完成(在RCX中保存 SYSCA
LL 之后指令地址之后)。
(WRMSR 指令确保IA32_LSTAR MSR总是包含一个连续的地址。)
...
...
...
SYSCALL 将 IA32_STAR MSR 的 47:32 位加载至 CS 和 SS 段选择器。
因此，根据这些段选择器 CS 和 SS ，描述符缓存并未从描述符加载(位于 GDT 或 LDT 中)。相反，描述符缓存
从固定值加载。
操作系统软件需要确保，由段选择器得到的描述符与从固定值加载至描述符缓存的描述符保持一致。 SYSCALL 指令
不保证两者的一致。
```

使用arch/x86/entry/entry_64.S汇编程序中定义的 `entry_SYSCALL_64` 初始化 `syscalls` 同时 `SYSCALL` 指令进入arch/x86/kernel/cpu/common.c 源码文件中的 `IA32_STAR` Model specific register:

```
wrmsrl(MSR_LSTAR, entry_SYSCALL_64);
```

因此， `syscall` 指令唤醒一个系统调用对应的处理程序。但是如何确定调用哪个处理程序？事实上这些信息从通用目的寄存器得到。正如系统调用表中描述，每个系统调用对应特定的编号。上面的示例中, 第一个系统调用是 - `write` 将数据写入指定文件。在系统调用表中查找 write 系统调用.write 系统调用的编号为 - `1` 。在示例中通过 `rax` 寄存器传递该编号，接下来的几个通用目的寄存器: `%rdi` , `%rsi` 和 `%rdx` 保存 `write` 系统调用的参数。 在示例中为文件描述符 (`1` 是stdout), 第二个参数字符串指针, 第三个为数据的大小。是的，你听到的没错，系统调用的参数。正如上文, 系统调用仅仅是内核空间的 `C` 函数。示例中第一个系统调用为 write ，在 [fs/read_write.c]
(https://github.com/torvalds/linux/blob/master/fs/read_write.c) 源文件中定义如下:

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
        size_t, count)
{
    ...
    ...
    ...
}
```

或者换言之:

```
ssize_t write(int fd, const void *buf, size_t nbytes);
```

现在不用担心宏 `SYSCALL_DEFINE3` ,稍后再做讨论。

示例的第二部分也是一样的, 但调用了另一系统调用exit。这个系统调用仅需一个参数:

- Return value

参数说明程序退出的方式。strace 工具可根据程序的名称输出系统调用的过程:

```
$ strace test
execve("./test", ["./test"], [/* 62 vars */]) = 0
write(1, "Hello, world!\n", 14Hello, world!
)          = 14
_exit(0)                                = ?

+++ exited with 0 +++
```

`strace` 输出的第一行, execve 系统调用开始执行程序，第二，三行为程序中使用的系统调用: `write` 和 `exit` 。注意示例中通过通用目的寄存器传递系统调用的参数。寄存器的顺序是特定的。寄存器的顺序由- 声明 [x86-64 calling conventions] (https://en.wikipedia.org/wiki/X86_calling_conventions#x86-64_calling_conventions)定义。 `x86_64` 架构的声明在另一个特别的文档中 - System V Application Binary Interface. PDF。通常, 函数参数被置于寄存器或者堆栈中。正确的顺序为:

- `rdi` ;
- `rsi` ;
- `rdx` ;
- `rcx` ;
- `r8` ;
- `r9` .

对应函数的前六个参数。若函数多于六个参数，其他参数将放在堆栈中。

示例代码中未直接使用系统调用，但程序通过系统调用打印输出，检查文件的权限或是从文件中读写。

例如:

```
#include <stdio.h>

int main(int argc, char **argv)
{
   FILE *fp;
   char buff[255];

   fp = fopen("test.txt", "r");
   fgets(buff, 255, fp);
   printf("%s\n", buff);
   fclose(fp);

   return 0;
}
```

Linux内核中没有 `fopen` , `fgets` , `printf` 和 `fclose` 系统调用，而是 `open` , `read` `write` 和 `close` 。 `fopen` , `fgets` , `printf` 和 `fclose` 仅仅是 `C` standard library中定义的函数。事实上这些函数是系统调用的封装。代码中没有直接使用系统调用，而是通过标准库的封装函数。这样做的主要原因是: 系统调用执行的要快，非常快。由于系统调用快的同时也非常小。标准库在执行系统调用前，确保系统调用参数设置正确及完成其他不同的检查。对比示例程序和以下命令:

```
$ gcc test.c -o test
```

通过ltrace工具观察:

```
$ ltrace ./test
__libc_start_main([ "./test" ] <unfinished ...>
fopen("test.txt", "r")                                          = 0x602010
fgets("Hello World!\n", 255, 0x602010)                          = 0x7ffd2745e700
puts("Hello World!\n"Hello World!

)                                                               = 14
fclose(0x602010)                                                = 0
+++ exited (status 0) +++
```

`ltrace` 工具显示程序用户空间的调用。 `fopen` 函数打开给定的文本文件, `fgets` 函数读取文件内容至 `buf` 缓存， `puts` 输出文件内容至 `stdout` ， `fclose` 函数根据文件描述符关闭函数。如上文描述，这些函数调用特定的系统调用。例如： `puts` 内部调用 `write` 系统调用， `ltrace` 添加 `-S` 可观察到这一调用:

```
write@SYS(1, "Hello World!\n\n", 14) = 14
```

系统调用是普遍存在的。每个程序都需要打开/写/读文件，网络连接，内存分配和许多其他功能只能由内核完成。proc 文件系统有一个具有特定格式的特殊文件: `/proc/pid/systemcall` 记录了正在被进程调用的系统调用的编号和参数寄存器。例如,进程号 1 的程序是systemd:

```
$ sudo cat /proc/1/comm
systemd

$ sudo cat /proc/1/syscall
232 0x4 0x7ffdf82e11b0 0x1f 0xffffffff 0x100 0x7ffdf82e11bf 0x7ffdf82e11a0 0x7f9114681
193
```

编号为 - `232` 的系统调用为 epoll_wait，该调用等待 epoll 文件描述符的I/O事件. 例如我用来编写这一节的 `emacs` 编辑器:

```
$ ps ax | grep emacs
2093 ?        Sl     2:40 emacs

$ sudo cat /proc/2093/comm
emacs

$ sudo cat /proc/2093/syscall
270 0xf 0x7fff068a5a90 0x7fff068a5b10 0x0 0x7fff068a59c0 0x7fff068a59d0 0x7fff068a59b0
 0x7f777dd8813c
```

编号为 `270` 的系统调用是 sys_pselect6 ，该系统调用使 `emacs` 监控多个文件描述符。

现在我们对系统调用有所了解，知道什么是系统调用及为什么需要系统调用。接下来，讨论示例程序中使用的 `write` 系统调用

写系统调用的实现

查看Linux内核源文件中写系统调用的实现。fs/read_write.c 源码文件中的 `write` 系统调用定义如下：

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
        size_t, count)
{
    struct fd f = fdget_pos(fd);
    ssize_t ret = -EBADF;

    if (f.file) {
        loff_t pos = file_pos_read(f.file);
        ret = vfs_write(f.file, buf, count, &pos);
        if (ret >= 0)
            file_pos_write(f.file, pos);
        fdput_pos(f);
    }

    return ret;
}
```

首先，宏 `SYSCALL_DEFINE3` 在头文件 include/linux/syscalls.h 中定义并且作为 `sys_name(...)` 函数定义的扩展。宏的定义如下:

```
#define SYSCALL_DEFINE3(name, ...) SYSCALL_DEFINEx(3, _##name, __VA_ARGS__)

#define SYSCALL_DEFINEx(x, sname, ...)                \
        SYSCALL_METADATA(sname, x, __VA_ARGS__)       \
        __SYSCALL_DEFINEx(x, sname, __VA_ARGS__)
```

宏 `SYSCALL_DEFINE3` 的参数有代表系统调用的名称的 `name` 和可变个数的参数。这个宏仅仅作为 `SYSCALL_DEFINEx` 宏的扩展确定了传入宏的参数个数。 `_##name` 作为未来系统调用名称的存根 (更多关于 `##` 符号连结可参阅documentation of gcc)。宏 `SYSCALL_DEFINEx` 作为以下两个宏的扩展:

- `SYSCALL_METADATA`;
- `__SYSCALL_DEFINEx`.

第一个宏 `SYSCALL_METADATA` 的实现与内核配置选项 `CONFIG_FTRACE_SYSCALLS` 有关。从选项的名称可知，选项允许 tracer 捕获系统调用的进入和退出。若该内核配置选项开启，宏 `SYSCALL_METADATA` 执行头文件include/trace/syscall.h 中 `syscall_metadata` 结构的初始化。结构中包含多种有用字段例如系统调用的名称, 系统调用表中的编号，参数个数, 参数类型列表等:

```
#define SYSCALL_METADATA(sname, nb, ...)                                \
    ...                                                                 \
    ...                                                                 \
    ...                                                                 \
    struct syscall_metadata __used                                      \
              __syscall_meta_##sname = {                                \
                    .name           = "sys"#sname,                      \
                    .syscall_nr     = -1,                               \
                    .nb_args        = nb,                               \
                    .types          = nb ? types_##sname : NULL,        \
                    .args           = nb ? args_##sname : NULL,         \
                    .enter_event    = &event_enter_##sname,             \
                    .exit_event     = &event_exit_##sname,              \
                    .enter_fields   = LIST_HEAD_INIT(__syscall_meta_##sname.enter_fiel
ds), \
             };
     \

    static struct syscall_metadata __used                              \
              __attribute__((section("__syscalls_metadata")))          \
             *__p_syscall_meta_##sname = &__syscall_meta_##sname;
```

若内核配置时 `CONFIG_FTRACE_SYSCALLS` 未开启，此时宏 `SYSCALL_METADATA` 扩展为空字符串:

```
#define SYSCALL_METADATA(sname, nb, ...)
```

第二个宏 `__SYSCALL_DEFINEx` 扩展为 以下五个函数的定义:

```
#define __SYSCALL_DEFINEx(x, name, ...)                                 \
        asmlinkage long sys##name(__MAP(x,__SC_DECL,__VA_ARGS__))       \
                __attribute__((alias(__stringify(SyS##name))));         \
                                                                        \
        static inline long SYSC##name(__MAP(x,__SC_DECL,__VA_ARGS__));  \
                                                                        \
        asmlinkage long SyS##name(__MAP(x,__SC_LONG,__VA_ARGS__));      \
                                                                        \
        asmlinkage long SyS##name(__MAP(x,__SC_LONG,__VA_ARGS__))       \
        {                                                               \
                long ret = SYSC##name(__MAP(x,__SC_CAST,__VA_ARGS__));  \
                __MAP(x,__SC_TEST,__VA_ARGS__);                         \
                __PROTECT(x, ret,__MAP(x,__SC_ARGS,__VA_ARGS__));       \
                return ret;                                             \
        }                                                               \
                                                                        \
        static inline long SYSC##name(__MAP(x,__SC_DECL,__VA_ARGS__))
```

第一个函数 `sys##name` 是给定名称 `sys_system_call_name` 系统调用处理器函数的定义。宏 `__SC_DECL` 的参数有 `__VA_ARGS__` 及组合调用传入参数系统类型和参数名称，因为宏定义中无法指定参数类型。宏 `__MAP` 应用宏 `__SC_DECL` 给 `__VA_ARGS__` 参数。其他由宏 `__SYSCALL_DEFINEx` 产生的函数需要 protect from the CVE-2009-0029 此处不必深入研究。作为宏 `SYSCALL_DEFINE3` 的结论:

```
asmlinkage long sys_write(unsigned int fd, const char __user * buf, size_t count);
```

现在我们对系统调用的定义有一定了解，回头讨论 `write` 系统调用的实现:

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
        size_t, count)
{
    struct fd f = fdget_pos(fd);
    ssize_t ret = -EBADF;

    if (f.file) {
        loff_t pos = file_pos_read(f.file);
        ret = vfs_write(f.file, buf, count, &pos);
        if (ret >= 0)
            file_pos_write(f.file, pos);
        fdput_pos(f);
    }

    return ret;
}
```

从代码可知，该调用有三个参数:

- `fd` - 文件描述符;
- `buf` - 写缓冲区;
- `count` - 写缓冲区大小.

调用的功能是将用户定义的缓冲中的数据写入指定的设备或文件。注意第二个参数 `buf`, 定义了 `__user` 属性。该属性的主要目的是通过 sparse 工具检查 Linux 内核代码。sparse 定义于 [include/linux/compiler.h](https://github.com/torvalds/linux/blob/master/include/linux/compiler.h) 头文件中并依赖 Linux 内核的 `__CHECKER__` 定义。其中全是关于 `sys_write` 系统调用的有用元信息。试着理解该系统调用的实现，定义从 `fd` 结构类型的 `f` 结构开始，这是 Linux 内核中的文件描述符。将调用的输出传入 `fdget_pos` 函数。 `fdget_pos` 函数在相同的源文件中定义，并且仅作为 `__to_fd` 函数的扩展:

```
static inline struct fd fdget_pos(int fd)
{
        return __to_fd(__fdget_pos(fd));
}
```

`fdget_pos` 的主要目的是将仅仅作为的数字的给定的文件描述符转化为 `fd` 结构。 通过一长链的函数调用， `fdget_pos` 函数得到当前进程的文件描述符表, `current->files` , 并尝试从表中获取一致的文件描述符编号。当获取到给定文件描述符的 `fd` 结构后, 检查文件并返回文件是否存在。通过调用函数 `file_pos_read` 获取当前处于文件中的位置。函数返回文件的 `f_pos` 字段:

```
static inline loff_t file_pos_read(struct file *file)
{
        return file->f_pos;
}
```

之后调用 `vfs_write` 函数。 `vfs_write` 函数在源码文件 fs/read_write.c 中定义。其功能为 - 向指定文件的指定位置写入指定缓冲中的数据。此处不深入 `vfs_write` 函数的细节，因为这个函数与 `系统调用` 没有太多联系，反而与另一章节Virtual file system相关。 `vfs_write` 结束相关工作后, 检查结果若成功执行，使用 `file_pos_write` 函数改变在文件中的位置:

```
if (ret >= 0)
    file_pos_write(f.file, pos);
```

这恰好使用给定的位置更新给定文件的 `f_pos` :

```
static inline void file_pos_write(struct file *file, loff_t pos)
{
        file->f_pos = pos;
}
```

在 `write` 系统调用处理函数的结束, 是以下函数:

```
fdput_pos(f);
```

解锁在共享文件描述符的线程并发写文件时保护文件位置的互斥量 `f_pos_lock` 。

我们讨论了Linux内核提供的系统调用的部分实现。显然略过了 `write` 系统调用的部分实现细节，正如文中所述, 在该章节中仅关心系统调用的相关内容，不讨论与其他子系统相关的内容，例如Virtual file system.

总结

总结Linux内核中关于系统调用概念的 the first part covering system call concepts in the Linux kernel. 本节中讨论了系统调用的原理，接下来的一节将深入该主题，了解 Linux 内核系统调用相关代码。

若存在疑问及建议, 在twitter @0xAX, 通过email 或者创建 issue.

由于英语是我的第一语言由此造成的不便深感抱歉。若发现错误请提交 **PR** 至 **linux-insides**.

链接

* system call
* vdso
* vsyscall
* general purpose registers
* socket
* C programming language
* x86
* x86_64
* x86-64 calling conventions
* System V Application Binary Interface. PDF
* GCC
* Intel manual. PDF
* system call table

- GCC macro documentation
- file descriptor
- stdout
- strace
- standard library
- wrapper functions
- ltrace
- sparse
- proc file system
- Virtual file system
- systemd
- epoll
- Previous chapter

Linux 系统内核调用 第二节

Linux 内核如何处理系统调用

前一小节 作为本章节的第一部分描述了 Linux 内核system call 概念。 前一节中提到通常系统调用处于内核处于操作系统层面。前一节内容从用户空间的角度介绍，并且 write系统调用实现的一部分内容没有讨论。在这一小节继续关注系统调用，在深入 Linux 内核之前，从一些理论开始。

程序中一个用户程序并不直接使用系统调用。我们并未这样写 `Hello World` 程序代码：

```
int main(int argc, char **argv)
{
    ...
    ...
    ...
    sys_write(fd1, buf, strlen(buf));
    ...
    ...
}
```

我们可以使用与 C standard library 帮助类似的方式:

```
#include <unistd.h>

int main(int argc, char **argv)
{
    ...
    ...
    ...
    write(fd1, buf, strlen(buf));
    ...
    ...
}
```

不管怎样， `write` 不是直接的系统调用也不是内核函数。程序必须将通用目的寄存器按照正确的顺序存入正确的值，之后使用 `syscall` 指令实现真正的系统调用。在这一节我们关注 Linux 内核中，处理器执行 `syscall` 指令时的细节。

系统调用表的初始化

从前一节可知系统调用与中断非常相似。深入的说，系统调用是软件中断的处理程序。因此，当处理器执行程序的 `syscall` 指令时，指令引起异常导致将控制权转移至异常处理。 众所周知，所有的异常处理 (或者内核 C 函数将响应异常) 是放在内核代码中的。但是 Linux 内核如何查找对应系统调用的系统调用处理程序的地址？ Linux 内核由一个特殊的表： `system call table` 。 系统调用表是Linux内核源码文件 arch/x86/entry/syscall_64.c 中定义的数组 `sys_call_table` 的对应。其实现如下:

```
asmlinkage const sys_call_ptr_t sys_call_table[__NR_syscall_max+1] = {
    [0 ... __NR_syscall_max] = &sys_ni_syscall,
    #include <asm/syscalls_64.h>
};
```

`sys_call_table` 数组的大小为 `__NR_syscall_max + 1` ， `__NR_syscall_max` 宏作为给定架构的系统调用最大数量。 这本书关于 x86_64 架构, 因此 `__NR_syscall_max` 为 `322` ，这也是本书编写时(当前 Linux 内核版本为 `4.2.0-rc8+`)的数字。编译内核时可通过 Kbuild产生的头文件查看该宏 - include/generated/asm-offsets.h`:

```
#define __NR_syscall_max 322
```

对于 `x86_64` ， arch/x86/entry/syscalls/syscall_64.tbl 中也有相同的系统调用数量。这里存在两个重要的话题; `sys_call_table` 数组的类型及数组中元数的初始值。首先， `sys_call_ptr_t` 为指向系统调用表的指针。 其是通过 [typedef] 定义的函数指针的 (https://en.wikipedia.org/wiki/Typedef) ，返回值为空且无参数：

```
typedef void (*sys_call_ptr_t)(void);
```

其次为 `sys_call_table` 数组中元素的初始化。从上面的代码中可知,数组中所有元素包含指向 `sys_ni_syscall` 的系统调用处理器的指针。 `sys_ni_syscall` 函数为 “not-implemented” 调用。 首先, `sys_call_table` 的所有元素指向 “not-implemented” 系统调用。这是正确的初始化方法，因为我们仅仅初始化指向系统调用处理器的指针的存储位置，稍后再做处理。 `sys_ni_syscall` 的结果比较简单, 仅仅返回 -errno 或者 `-ENOSYS` ：

```
asmlinkage long sys_ni_syscall(void)
{
    return -ENOSYS;
}
```

The `-ENOSYS` error tells us that:

```
ENOSYS          Function not implemented (POSIX.1)
```

在 `sys_call_table` 的初始化中同时也要注意 `...` 。可通过 GCC 编译器插件 - Designated Initializers 处理。插件允许使用不固定的顺序初始化元素。 在数组结束处，我们引用 `asm/syscalls_64.h` 头文件在。头文件由特殊的脚本 arch/x86/entry/syscalls/syscalltbl.sh 从 syscall table 产生。 `asm/syscalls_64.h` 包括以下宏的定义:

```
__SYSCALL_COMMON(0, sys_read, sys_read)
__SYSCALL_COMMON(1, sys_write, sys_write)
__SYSCALL_COMMON(2, sys_open, sys_open)
__SYSCALL_COMMON(3, sys_close, sys_close)
__SYSCALL_COMMON(5, sys_newfstat, sys_newfstat)
...
...
...
```

宏 `__SYSCALL_COMMON` 在相同的 源码 中定义，作为宏 `__SYSCALL_64` 的扩展:

```
#define __SYSCALL_COMMON(nr, sym, compat) __SYSCALL_64(nr, sym, compat)
#define __SYSCALL_64(nr, sym, compat) [nr] = sym,
```

因而, 到此为止, `sys_call_table` 为如下格式:

```
asmlinkage const sys_call_ptr_t sys_call_table[__NR_syscall_max+1] = {
    [0 ... __NR_syscall_max] = &sys_ni_syscall,
    [0] = sys_read,
    [1] = sys_write,
    [2] = sys_open,
    ...
    ...
    ...
};
```

之后所有指向“ non-implemented ”系统调用元素的内容为 `sys_ni_syscall` 函数的地址，该函数仅返回 `-ENOSYS` 。 其他元素指向 `sys_syscall_name` 函数。

至此, 完成系统调用表的填充并且 Linux内核了解系统调用处理器的为值。但是 Linux 内核在处理用户空间程序的系统调用时并未立即调用 `sys_syscall_name` 函数。 记住关于中断及中断处理的 章节 。当 Linux 内核获得处理中断的控制权, 在调用中断处理程序前，必须做一些准备如保存用户空间寄存器，切换至新的堆栈及其他很多工作。系统调用处理也是相同的情形。第一件事是处理系统调用的准备，但是在 Linux 内核开始这些准备之前, 系统调用的入口必须完成初始化，同时只有 Linux 内核知道如何执行这些准备。在下一章节我们将关注 Linux 内核中关于系统调用入口的初始化过程。

系统调用入口初始化

当系统中发生系统调用，开始处理调用的代码的第一个字节在什么地方？阅读 Intel 的手册 - 64-ia-32-architectures-software-developer-vol-2b-manual:

```
SYSCALL 引起操作系统系统调用处理器处于特权级0，通过加载IA32_LSTAR MSR至RIP完成。
```

这就是说我们需要将系统调用入口放置到 `IA32_LSTAR` model specific register 。 这一操作在 Linux 内核初始过程时完成。若已阅读关于 Linux 内核中断及中断处理政界的 第四节 , Linux 内核调用在初始化过程中调用 `trap_init` 函数。该函数在 arch/x86/kernel/setup.c 源代码文件中定义，执行 `non-early` 异常处理（如除法错误，协处理器 错误等 ）的初始化。除了 `non-early` 异常处理的初始化外，函数调用 arch/x86/kernel/cpu/common.c 中 `cpu_init` 函数，调用相同源码文件中的 `syscall_init` 完成 `per-cpu` 状态初始化。

该函数执行系统调用入口的初始化。查看函数的实现，函数没有参数且首先填充两个特殊模块寄存器：

```
wrmsrl(MSR_STAR,  ((u64)__USER32_CS)<<48  | ((u64)__KERNEL_CS)<<32);
wrmsrl(MSR_LSTAR, entry_SYSCALL_64);
```

第一个特殊模块集寄存器- `MSR_STAR` 的 `63:48` 为用户代码的代码段。这些数据将加载至 `CS` 和 `SS` 段选择符，由提供将系统调用返回至相应特权级的用户代码功能的 `sysret` 指令使用。 同时从内核代码来看， 当用户空间应用程序执行系统调用时， `MSR_STAR` 的 `47:32` 将作为 `CS` and `SS` 段选择寄存器的基地址。第二行代码中我们将使用系统调用入口 `entry_SYSCALL_64` 填充 `MSR_LSTAR` 寄存器。 `entry_SYSCALL_64` 在 arch/x86/entry/entry_64.S 汇编文件中定义，包含系统调用执行前的准备(上面已经提及这些准备)。 目前不关注 `entry_SYSCALL_64` ,将在章节的后续讨论。

在设置系统调用的入口之后，需要以下特殊模式寄存器：

- `MSR_CSTAR` - target `rip` for the compability mode callers;
- `MSR_IA32_SYSENTER_CS` - target `cs` for the `sysenter` instruction;
- `MSR_IA32_SYSENTER_ESP` - target `esp` for the `sysenter` instruction;
- `MSR_IA32_SYSENTER_EIP` - target `eip` for the `sysenter` instruction.

这些特殊模式寄存器的值与内核配置选项 `CONFIG_IA32_EMULATION` 有关。 若开启该内核配置选项，允许64字节内核运行32字节的程序。 首先, 若 `CONFIG_IA32_EMULATION` 内合配置选项开启, 将使用兼容模式的系统调用入口填充这些特殊模式寄存器：

```
wrmsrl(MSR_CSTAR, entry_SYSCALL_compat);
```

对于内核代码段, 将堆栈指针置零， `entry_SYSENTER_compat` 字的地址写入指令指针:

```
wrmsrl_safe(MSR_IA32_SYSENTER_CS, (u64)__KERNEL_CS);
wrmsrl_safe(MSR_IA32_SYSENTER_ESP, 0ULL);
wrmsrl_safe(MSR_IA32_SYSENTER_EIP, (u64)entry_SYSENTER_compat);
```

另一方面，若 `CONFIG_IA32_EMULATION` 内核配置选项未开启，将把 `ignore_sysret` 字写入 `MSR_CSTAR`：

```
wrmsrl(MSR_CSTAR, ignore_sysret);
```

其在arch/x86/entry/entry_64.S 汇编文件中定义，仅返回 `-ENOSYS` 错误代码：

```
ENTRY(ignore_sysret)
    mov     $-ENOSYS, %eax
    sysret
END(ignore_sysret)
```

现在需要像之前代码一样填充 `MSR_IA32_SYSENTER_CS`，`MSR_IA32_SYSENTER_ESP`，`MSR_IA32_SYSENTER_EIP` 特殊模式寄存器，当 `CONFIG_IA32_EMULATION` 内核配置选项打开时。在这种情况(`CONFIG_IA32_EMULATION` 配置选项未设置) 将用零填充 `MSR_IA32_SYSENTER_ESP` 和 `MSR_IA32_SYSENTER_EIP` ，同时将 Global Descriptor Table 的无效段加载至 `MSR_IA32_SYSENTER_CS` 特殊模式寄存器：

```
wrmsrl_safe(MSR_IA32_SYSENTER_CS, (u64)GDT_ENTRY_INVALID_SEG);
wrmsrl_safe(MSR_IA32_SYSENTER_ESP, 0ULL);
wrmsrl_safe(MSR_IA32_SYSENTER_EIP, 0ULL);
```

可以从描述 Linux 内核启动过程的章节阅读更多关于 `Global Descriptor Table` 的内容。

在 `syscall_init` 函数的结束，通过写入 `MSR_SYSCALL_MASK` 特殊寄存器的标志位，将 标志寄存器 中的标志位屏蔽：

```
wrmsrl(MSR_SYSCALL_MASK,
       X86_EFLAGS_TF|X86_EFLAGS_DF|X86_EFLAGS_IF|
       X86_EFLAGS_IOPL|X86_EFLAGS_AC|X86_EFLAGS_NT);
```

这些标志位将在 syscall 初始化时清除。至此，`syscall_init` 函数结束 也意味着系统调用已经可用。现在我们关注当用户程序执行 `syscall` 指令发生什么。

系统调用处理执行前的准备

如之前写到，系统调用或中断处理在被 Linux 内核调用前需要一些准备。宏 `idtentry` 完成异常处理被执行前的所需准备，宏 `interrupt` 完成中断处理被调用前的所需准备，`entry_SYSCALL_64` 完成系统调用执行前的所需准备。

`entry_SYSCALL_64` 在 arch/x86/entry/entry_64.S 汇编文件中定义 ，从下面的宏开始:

```
SWAPGS_UNSAFE_STACK
```

该宏在 arch/x86/include/asm/irqflags.h 头文件中定义， 扩展 `swapgs` 指令:

```
#define SWAPGS_UNSAFE_STACK     swapgs
```

宏将交换 GS 段选择符及 `MSR_KERNEL_GS_BASE` 特殊模式寄存器中的值。换句话说，将其入内核堆栈 。之后使老的堆栈指针指向 `rsp_scratch` per-cpu 变量设置堆栈指针指向当前处理器的栈顶：

```
movq    %rsp, PER_CPU_VAR(rsp_scratch)
movq    PER_CPU_VAR(cpu_current_top_of_stack), %rsp
```

下一步中将堆栈段及老的堆栈指针如栈：

```
pushq    $__USER_DS
pushq    PER_CPU_VAR(rsp_scratch)
```

之后使能中断，因为入口中断被关闭，保存通用目的 寄存器 (除 `bp` , `bx` 及 `r12` 至 `r15`)，标志位，“ non-implemented ” 系统调用相关的 `-ENOSYS` 及代码段寄存器至堆栈:

```
ENABLE_INTERRUPTS(CLBR_NONE)

pushq    %r11
pushq    $__USER_CS
pushq    %rcx
pushq    %rax
pushq    %rdi
pushq    %rsi
pushq    %rdx
pushq    %rcx
pushq    $-ENOSYS
pushq    %r8
pushq    %r9
pushq    %r10
pushq    %r11
sub    $(6*8), %rsp
```

当系统调用由用户空间程序引起时, 通用目的寄存器状态如下:

- `rax` - contains system call number;
- `rcx` - contains return address to the user space;
- `r11` - contains register flags;
- `rdi` - contains first argument of a system call handler;
- `rsi` - contains second argument of a system call handler;
- `rdx` - contains third argument of a system call handler;
- `r10` - contains fourth argument of a system call handler;
- `r8` - contains fifth argument of a system call handler;
- `r9` - contains sixth argument of a system call handler;

其他通用目的寄存器 (如 `rbp` , `rbx` 和 `r12` 至 `r15`) 在C ABI)保留。将寄存器标志位如栈，之后是 “non-implemented ”系统调用的用户代码段，用户空间返回地址，系统调用编号，三个参数，dump 错误代码和堆栈中的其他信息。

下一步检查当前 `thread_info` 中的 `_TIF_WORK_SYSCALL_ENTRY` :

```
testl    $_TIF_WORK_SYSCALL_ENTRY, ASM_THREAD_INFO(TI_flags, %rsp, SIZEOF_PTREGS)
jnz     tracesys
```

宏 `_TIF_WORK_SYSCALL_ENTRY` 在 arch/x86/include/asm/thread_info.h 头文件中定义 ，提供一系列与系统调用跟踪有关的进程信息标志:

```
#define _TIF_WORK_SYSCALL_ENTRY \
    (_TIF_SYSCALL_TRACE | _TIF_SYSCALL_EMU | _TIF_SYSCALL_AUDIT |    \
    _TIF_SECCOMP | _TIF_SINGLESTEP | _TIF_SYSCALL_TRACEPOINT |      \
    _TIF_NOHZ)
```

本章节中不讨论追踪/调试相关内容,将在关于 Linux 内核调试及追踪相关独立章节中讨论。 在 `tracesys` 标签之后, 下一标签为 `entry_SYSCALL_64_fastpath` .在 `entry_SYSCALL_64_fastpath` 中检查 头文件 arch/x86/include/asm/unistd.h 中定义的 `__SYSCALL_MASK`

```
# ifdef CONFIG_X86_X32_ABI
#  define __SYSCALL_MASK (~(__X32_SYSCALL_BIT))
# else
#  define __SYSCALL_MASK (~0)
# endif
```

`__X32_SYSCALL_BIT` 为 :

```
#define __X32_SYSCALL_BIT    0x40000000
```

众所周知， `__SYSCALL_MASK` 与 `CONFIG_X86_X32_ABI` 内核配置选项相关， 作为 64位内核中 32位ABI 的掩码。

So we check the value of the `__SYSCALL_MASK` and if the `CONFIG_X86_X32_ABI` is disabled we compare the value of the `rax` register to the maximum syscall number (`__NR_syscall_max`), alternatively if the `CNOFIG_X86_X32_ABI` is enabled we mask the `eax` register with the `__X32_SYSCALL_BIT` and do the same comparison:

```
#if __SYSCALL_MASK == ~0
    cmpq    $__NR_syscall_max, %rax
#else
    andl    $__SYSCALL_MASK, %eax
    cmpl    $__NR_syscall_max, %eax
#endif
```

至此检查最后一调比较指令的结果， `ja` 指令在 `CF` 和 `ZF` 标志为 0 时执行:

```
ja    1f
```

若正确调用系统调用, 从 `r10` 移动第四个参数至 `rcx` ，保持 x86_64 C ABI 开启，同时以系统调用的处理程序的地址为参数执行 `call` 指令:

```
movq    %r10, %rcx
call    *sys_call_table(, %rax, 8)
```

注意, 上文提到 `sys_call_table` 是一个数组。 `rax` 通用目的寄存器为系统调用的编号，且 `sys_call_table` 的每个元素为 8 字节。 因此使用 `*sys_call_table(, %rax, 8)` 符号找到指定系统调用处理在 `sys_call_table` 中的偏移。

就这样。完成了所需的准备，系统调用处理将被相应的中断处理调用。 例如 Linux 内核代码中 `SYSCALL_DEFINE[N]` 宏定义的 `sys_read` , `sys_write` 和其他中断处理。

退出系统调用

在系统调用处理完成人物后, 将退回arch/x86/entry/entry_64.S, 正好在系统调用之后:

```
call    *sys_call_table(, %rax, 8)
```

在从系统调用处理返回之后，下一步是将系统调用处理的返回值入栈。系统调用将用户程序的返回结果放置在通用目的寄存器 `rax` 中,因此在系统调用处理完成其工作后，将寄存器的值入栈：

```
movq    %rax, RAX(%rsp)
```

在 `RAX` 指定的位置。

之后调用在 arch/x86/include/asm/irqflags.h 中定义的宏 `LOCKDEP_SYS_EXIT` :

```
LOCKDEP_SYS_EXIT
```

宏的实现与 `CONFIG_DEBUG_LOCK_ALLOC` 内核配置选项相关，该配置允许在退出系统调用时调试锁。再次强调，在该章节不关注，将在单独的章节讨论相关内容。 在 `entry_SYSCALL_64` 函数的最后， 恢复除 `rxc` 和 `r11` 外所有通用寄存器, 因为 `rcx` 寄存器为调用系统调用的应用程序的返回地址， `r11` 寄存器为老的 flags register. 在恢复所有通用寄存器之后， 将在 `rcx` 中装入返回地址, `r11` 寄存器装入标志 ， `rsp` 装入老的堆栈指针:

```
RESTORE_C_REGS_EXCEPT_RCX_R11

movq    RIP(%rsp), %rcx
movq    EFLAGS(%rsp), %r11
movq    RSP(%rsp), %rsp

USERGS_SYSRET64
```

最后仅仅调用宏 `USERGS_SYSRET64` ，其扩展调用 `swapgs` 指令交换用户 `GS` 和内核 `GS` ， `sysretq` 指令执行从系统调用处理退出。

```
#define USERGS_SYSRET64                 \
    swapgs;                             \
    sysretq;
```

现在我们知道，当用户程序使用系统调用时发生的一切。整个过程的步骤如下：

- 用户程序中的代码装入通用目的寄存器的值（系统调用编号和系统调用的参数）;
- 处理器从用户模式切换到内核模式 开始执行系统调用入口 - `entry_SYSCALL_64` ;
- `entry_SYSCALL_64` 切换至内核堆栈，在堆栈中存通用目的寄存器, 老的堆栈，代码段, 标志位等;
- `entry_SYSCALL_64` 检查 `rax` 寄存器中的系统调用编号,系统调用编号正确时， 在 `sys_call_table` 中查找系统调用处理并调用;
- 若系统调用编号不正确, 跳至系统调用退出;
- 系统调用处理完成工作后, 恢复通用寄存器, 老的堆栈，标志位 及返回地址 ，通过 `sysretq` 指令退出 `entry_SYSCALL_64` .

结论

这是 Linux 内核相关概念的第二节。在前一 节 ，从用户应用程序的角度讨论了这些概念的原理。在这一节继续深入系统调用概念的相关内容，讨论了系统调用发生时 Linux 内核执行的内容。

若存在疑问及建议, 在twitter @0xAX, 通过email 或者创建 issue.

由于英语是我的第一语言由此造成的不便深感抱歉。若发现错误请提交 **PR** 至 **linux-insides**.

Links

- system call
- write
- C standard library
- list of cpu architectures
- x86_64
- kbuild
- typedef
- errno
- gcc
- model specific register
- intel 2b manual
- coprocessor
- instruction pointer
- flags register
- Global Descriptor Table
- per-cpu
- general purpose registers
- ABI
- x86_64 C ABI
- previous chapter

Linux 内核系统调用 第三节

vsyscalls 和 vDSO

这是讲解 Linux 内核中系统调用章节的第三部分，前一节讨论了用户空间应用程序发起的系统调用的准备工作及系统调用的处理过程。在这一节将讨论两个与系统调用十分相似的概念，这两个概念是 `vsyscall` 和 `vdso` 。

我们已经了解什么是 `系统调用` 。这是 Linux 内核一种特殊的运行机制，使得用户空间的应用程序可以请求，像写入文件和打开套接字等特权级下的任务。正如你所了解的，在 Linux 内核中发起一个系统调用是特别昂贵的操作，因为处理器需要中断当前正在执行的任务，切换内核模式的上下文，在系统调用处理完毕后跳转至用户空间。以下的两种机制 - `vsyscall` 和d `vdso` 被设计用来加速系统调用的处理，在这一节我们将了解两种机制的工作原理。

vsyscalls 介绍

`vsyscall` 或 `virtual system call` 是第一种也是最古老的一种用于加快系统调用的机制。`vsyscall` 的工作原则其实十分简单。Linux 内核在用户空间映射一个包含一些变量及一些系统调用的实现的内存页。 对于 X86_64 架构可以在 Linux 内核的 [文档](https://github.com/torvalds/linux/blob/master/Documentation/x86/x86_64/mm.txt) 找到关于这一内存区域的信息：

```
ffffffffff600000 - ffffffffffdfffff (=8 MB) vsyscalls
```

或:

```
~$ sudo cat /proc/1/maps | grep vsyscall
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

因此, 这些系统调用将在用户空间下执行，这意味着将不发生 上下文切换。 `vsyscall` 内存页的映射在 arch/x86/entry/vsyscall/vsyscall_64.c 源代码中定义的 `map_vsyscall` 函数中实现。这一函数在 Linux 内核初始化时被 arch/x86/kernel/setup.c 源代码中定义的函数 `setup_arch` (我们在第五章 Linux 内核的初始化中讨论过该函数)。

注意 `map_vsyscall` 函数的实现依赖于内核配置选项 `CONFIG_X86_VSYSCALL_EMULATION` :

```
#ifdef CONFIG_X86_VSYSCALL_EMULATION
extern void map_vsyscall(void);
#else
static inline void map_vsyscall(void) {}
#endif
```

正如帮助文档中所描述的, `CONFIG_X86_VSYSCALL_EMULATION` 配置选项: `使能 vsyscall 模拟`. 为何模拟 `vsyscall` ? 事实上, `vsyscall` 由于安全原因是一种遗留 ABI 。虚拟系统调用具有绑定的地址, 意味着 `vsyscall` 的内存页的位置在任何时刻是相同，这一位置是在 `map_vsyscall` 函数中指定的。这一函数的实现如下:

```
void __init map_vsyscall(void)
{
    extern char __vsyscall_page;
    unsigned long physaddr_vsyscall = __pa_symbol(&__vsyscall_page);
    ...
    ...
    ...
}
```

在 `map_vsyscall` 函数的开始，通过宏 `__pa_symbol` 获取了 `vsyscall` 内存页的物理地址(我们已在第四章 of the Linux kernel initialization process)讨论了该宏的实现）。 `__vsyscall_page` 在 arch/x86/entry/vsyscall/vsyscall_emu_64.S 汇编源代码文件中定义， 具有如下的 虚拟地址:

```
ffffffff81881000 D __vsyscall_page
```

在 `.data..page_aligned, aw` 段 中 包含如下三中系统调用:

- `gettimeofday` ;
- `time` ;
- `getcpu` .

或:

```
__vsyscall_page:
    mov $__NR_gettimeofday, %rax
    syscall
    ret

    .balign 1024, 0xcc
    mov $__NR_time, %rax
    syscall
    ret

    .balign 1024, 0xcc
    mov $__NR_getcpu, %rax
    syscall
    ret
```

回到 `map_vsyscall` 函数及 `__vsyscall_page` 的实现，在得到 `__vsyscall_page` 的物理地址之后，使用 `__set_fixmap` 为 `vsyscall` 内存页 检查设置 fix-mapped地址的变量 `vsyscall_mode`：

```
if (vsyscall_mode != NONE)
    __set_fixmap(VSYSCALL_PAGE, physaddr_vsyscall,
                 vsyscall_mode == NATIVE
                             ? PAGE_KERNEL_VSYSCALL
                             : PAGE_KERNEL_VVAR);
```

The `__set_fixmap` takes three arguments: The first is index of the `fixed_addresses` enum. In our case `VSYSCALL_PAGE` is the first element of the `fixed_addresses` enum for the `x86_64` architecture:

```
enum fixed_addresses {
...
...
...
#ifdef CONFIG_X86_VSYSCALL_EMULATION
    VSYSCALL_PAGE = (FIXADDR_TOP - VSYSCALL_ADDR) >> PAGE_SHIFT,
#endif
...
...
...
```

该变量值为 `511` 。第二个参数为映射内存页的物理地址，第三个参数为内存页的标志位。注意 `VSYSCALL_PAGE` 标志位依赖于变量 `vsyscall_mode` 。当 `vsyscall_mode` 变量为 `NATIVE` 时， 标志位为 `PAGE_KERNEL_VSYSCALL` ，其他情况则是 `PAGE_KERNEL_VVAR` 。两个宏 (`PAGE_KERNEL_VSYSCALL` 及 `PAGE_KERNEL_VVAR`) 都将被扩展以下标志:

```
#define __PAGE_KERNEL_VSYSCALL          (__PAGE_KERNEL_RX | _PAGE_USER)
#define __PAGE_KERNEL_VVAR              (__PAGE_KERNEL_RO | _PAGE_USER)
```

标志反映了 `vsyscall` 内存页的访问权限。两个标志都带有 `_PAGE_USER` 标志，这意味着内存页可被运行于低特权级的用户模式进程访问。第二个标志位取决于 `vsyscall_mode` 变量的值。第一个标志 (`__PAGE_KERNEL_VSYSCALL`) 在 `vsyscall_mode` 为 `NATIVE` 时被设定。这意味着虚拟系统调用将以本地 `syscall` 指令的方式执行。另一情况下，在 `vsyscall_mode` 为 `emulate` 时 vsyscall 为 `PAGE_KERNEL_VVAR` ，此时系统调用将被置于陷阱并被合理的模拟。`vsyscall_mode` 变量通过 `vsyscall_setup` 获取值：

```
static int __init vsyscall_setup(char *str)
{
    if (str) {
        if (!strcmp("emulate", str))
            vsyscall_mode = EMULATE;
        else if (!strcmp("native", str))
            vsyscall_mode = NATIVE;
        else if (!strcmp("none", str))
            vsyscall_mode = NONE;
        else
            return -EINVAL;

        return 0;
    }

    return -EINVAL;
}
```

函数将在早期的内核分析时被调用：

```
early_param("vsyscall", vsyscall_setup);
```

关于 `early_param` 宏的更多信息可以在第六章 Linux 内核初始化中找到。

在函数 `vsyscall_map` 的最后仅通过 BUILD_BUG_ON 宏检查 `vsyscall` 内存页的虚拟地址是否等于变量 `VSYSCALL_ADDR` ：

```
BUILD_BUG_ON((unsigned long)__fix_to_virt(VSYSCALL_PAGE) !=
             (unsigned long)VSYSCALL_ADDR);
```

就这样 `vsyscall` 内存页设置完毕。上述的结果如下：若设置 `vsyscall=native` 内核命令行参数，虚拟内存调用将以 arch/x86/entry/vsyscall/vsyscall_emu_64.S 文件中本地 `系统调用` 指令的方式执行。 glibc 知道虚拟系统调用处理器的地址。注意虚拟系统调用的地址以 `1024` (或 `0x400`) 比特对齐。

```
__vsyscall_page:
    mov $__NR_gettimeofday, %rax
    syscall
    ret

    .balign 1024, 0xcc
    mov $__NR_time, %rax
    syscall
    ret

    .balign 1024, 0xcc
    mov $__NR_getcpu, %rax
    syscall
    ret
```

`vsyscall` 内存页的起始地址为 `ffffffffff600000` 。因此, glibc 知道所有虚拟系统调用处理器的地址。可以在 `glibc` 源码中找到这些地址的定义：

```
#define VSYSCALL_ADDR_vgettimeofday   0xffffffffff600000
#define VSYSCALL_ADDR_vtime           0xffffffffff600400
#define VSYSCALL_ADDR_vgetcpu          0xffffffffff600800
```

所有的虚拟系统调用请求都将映射至 `__vsyscall_page` + `VSYSCALL_ADDR_vsyscall_name` 偏置, 将虚拟内存系统调用的编号置于通用目的寄存器，本地的 x86_64 `系统调用` 指令将被执行。

在第二种情况中, 若将 `vsyscall=emulate` 参数传递给内核命令行, 提升虚拟系统调用处理器的尝试导致一个 page fault 异常。 谨记, `vsyscall` 内存页 具有 `__PAGE_KERNEL_VVAR` 的访问权限，这将禁止执行。 `do_page_fault` 函数是 `#PF` 或 page fault 的处理器。它将尝试了解最后一次 page fault 的原因。一种可能的场景是 `vsyscall` 模式为 `emulate` 情况下的虚拟系统调用。此时 `vsyscall` 将被 arch/x86/entry/vsyscall/vsyscall_64.c 源码中定义的 `emulate_vsyscall` 函数处理。

The `emulate_vsyscall` function gets the number of a virtual system call, checks it, prints error and sends segementation fault single:

```
...
...
...
vsyscall_nr = addr_to_vsyscall_nr(address);
if (vsyscall_nr < 0) {
    warn_bad_vsyscall(KERN_WARNING, regs, "misaligned vsyscall...);
    goto sigsegv;
}
...
...
...
sigsegv:
    force_sig(SIGSEGV, current);
    reutrn true;
```

As it checked number of a virtual system call, it does some yet another checks like `access_ok` violations and execute system call function depends on the number of a virtual system call:

```
switch (vsyscall_nr) {
    case 0:
        ret = sys_gettimeofday(
            (struct timeval __user *)regs->di,
            (struct timezone __user *)regs->si);
        break;
    ...
    ...
    ...
}
```

In the end we put the result of the `sys_gettimeofday` or another virtual system call handler to the `ax` general purpose register, as we did it with the normal system calls and restore the instruction pointer register and add `8` bytes to the stack pointer register. This operation emulates `ret` instruction.

```
    regs->ax = ret;

do_ret:
    regs->ip = caller;
    regs->sp += 8;
    return true;
```

That's all. Now let's look on the modern concept - `vDSO`.

Introduction to vDSO

As I already wrote above, `vsyscall` is an obsolete concept and replaced by the `vDSO` or `virtual dynamic shared object`. The main difference between the `vsyscall` and `vDSO` mechanisms is that `vDSO` maps memory pages into each process in a shared object form, but `vsyscall` is static in memory and has the same address every time. For the `x86_64` architecture it is called - `linux-vdso.so.1`. All userspace applications linked with this shared library via the `glibc`. For example:

```
~$ ldd /bin/uname
    linux-vdso.so.1 (0x00007ffe014b7000)
    libc.so.6 => /lib64/libc.so.6 (0x00007fbfee2fe000)
    /lib64/ld-linux-x86-64.so.2 (0x00005559aab7c000)
```

Or:

```
~$ sudo cat /proc/1/maps | grep vdso
7fff39f73000-7fff39f75000 r-xp 00000000 00:00 0       [vdso]
```

Here we can see that uname util was linked with the three libraries:

- `linux-vdso.so.1`;
- `libc.so.6`;
- `ld-linux-x86-64.so.2`.

The first provides `vDSO` functionality, the second is `C` standard library and the third is the program interpreter (more about this you can read in the part that describes linkers). So, the `vDSO` solves limitations of the `vsyscall`. Implementation of the `vDSO` is similar to `vsyscall`.

Initialization of the `vDSO` occurs in the `init_vdso` function that defined in the arch/x86/entry/vdso/vma.c source code file. This function starts from the initialization of the `vDSO` images for 32-bits and 64-bits depends on the `CONFIG_X86_X32_ABI` kernel configuration option:

```
static int __init init_vdso(void)
{
    init_vdso_image(&vdso_image_64);

#ifdef CONFIG_X86_X32_ABI
    init_vdso_image(&vdso_image_x32);
#endif
```

Both function initialize the `vdso_image` structure. This structure is defined in the two generated source code files: the arch/x86/entry/vdso/vdso-image-64.c and the arch/x86/entry/vdso/vdso-image-64.c. These source code files generated by the vdso2c

program from the different source code files, represent different approaches to call a system call like `int 0x80`, `sysenter` and etc. The full set of the images depends on the kernel configuration.

For example for the `x86_64` Linux kernel it will contain `vdso_image_64`:

```
#ifdef CONFIG_X86_64
extern const struct vdso_image vdso_image_64;
#endif
```

But for the `x86` - `vdso_image_32`:

```
#ifdef CONFIG_X86_X32
extern const struct vdso_image vdso_image_x32;
#endif
```

If our kernel is configured for the `x86` architecture or for the `x86_64` and compability mode, we will have ability to call a system call with the `int 0x80` interrupt, if compability mode is enabled, we will be able to call a system call with the native `syscall instruction` or `sysenter` instruction in other way:

```
#if defined CONFIG_X86_32 || defined CONFIG_COMPAT
  extern const struct vdso_image vdso_image_32_int80;
#ifdef CONFIG_COMPAT
  extern const struct vdso_image vdso_image_32_syscall;
#endif
 extern const struct vdso_image vdso_image_32_sysenter;
#endif
```

As we can understand from the name of the `vdso_image` structure, it represents image of the `vDSO` for the certain mode of the system call entry. This structure contains information about size in bytes of the `vDSO` area that always a multiple of `PAGE_SIZE` (`4096` bytes), pointer to the text mapping, start and end address of the `alternatives` (set of instructions with better alternatives for the certain type of the processor) and etc. For example `vdso_image_64` looks like this:

```
const struct vdso_image vdso_image_64 = {
    .data = raw_data,
    .size = 8192,
    .text_mapping = {
        .name = "[vdso]",
        .pages = pages,
    },
    .alt = 3145,
    .alt_len = 26,
    .sym_vvar_start = -8192,
    .sym_vvar_page = -8192,
    .sym_hpet_page = -4096,
};
```

Where the `raw_data` contains raw binary code of the 64-bit `vDSO` system calls which are `2` page size:

```
static struct page *pages[2];
```

or 8 Kilobytes.

The `init_vdso_image` function is defined in the same source code file and just initializes the `vdso_image.text_mapping.pages`. First of all this function calculates the number of pages and initializes each `vdso_image.text_mapping.pages[number_of_page]` with the `virt_to_page` macro that converts given address to the `page` structure:

```
void __init init_vdso_image(const struct vdso_image *image)
{
    int i;
    int npages = (image->size) / PAGE_SIZE;

    for (i = 0; i < npages; i++)
        image->text_mapping.pages[i] =
            virt_to_page(image->data + i*PAGE_SIZE);
    ...
    ...
    ...
}
```

The `init_vdso` function passed to the `subsys_initcall` macro adds the given function to the `initcalls` list. All functions from this list will be called in the `do_initcalls` function from the init/main.c source code file:

```
subsys_initcall(init_vdso);
```

Ok, we just saw initialization of the `vDSO` and initialization of `page` structures that are related to the memory pages that contain `vDSO` system calls. But to where do their pages map? Actually they are mapped by the kernel, when it loads binary to the memory. The Linux kernel calls the `arch_setup_additional_pages` function from the arch/x86/entry/vdso/vma.c source code file that checks that `vDSO` enabled for the `x86_64` and calls the `map_vdso` function:

```
int arch_setup_additional_pages(struct linux_binprm *bprm, int uses_interp)
{
    if (!vdso64_enabled)
        return 0;

    return map_vdso(&vdso_image_64, true);
}
```

The `map_vdso` function is defined in the same source code file and maps pages for the `vDSO` and for the shared `vDSO` variables. That's all. The main differences between the `vsyscall` and the `vDSO` concepts is that `vsyscal` has a static address of `ffffffffff600000` and implements `3` system calls, whereas the `vDSO` loads dynamically and implements four system calls:

- `__vdso_clock_gettime` ;
- `__vdso_getcpu` ;
- `__vdso_gettimeofday` ;
- `__vdso_time` .

That's all.

Conclusion

This is the end of the third part about the system calls concept in the Linux kernel. In the previous part we discussed the implementation of the preparation from the Linux kernel side, before a system call will be handled and implementation of the `exit` process from a system call handler. In this part we continued to dive into the stuff which is related to the system call concept and learned two new concepts that are very similar to the system call - the `vsyscall` and the `vDSO` .

After all of these three parts, we know almost all things that are related to system calls, we know what system call is and why user applications need them. We also know what occurs when a user application calls a system call and how the kernel handles system calls.

The next part will be the last part in this chapter and we will see what occurs when a user runs the program.

If you have questions or suggestions, feel free to ping me in twitter 0xAX, drop me email or just create issue.

Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-insides.

Links

- x86_64 memory map
- x86_64
- context switching
- ABI
- virtual address
- Segmentation
- enum
- fix-mapped addresses
- glibc
- BUILD_BUG_ON
- Processor register
- Page fault
- segementation fault
- instruction pointer
- stack pointer
- uname
- Linkers
- Previous part

System calls in the Linux kernel. Part 4.

How does the Linux kernel run a program

This is the fourth part of the chapter that describes system calls in the Linux kernel and as I wrote in the conclusion of the previous - this part will be last in this chapter. In the previous part we stopped at the two new concepts:

* `vsyscall` ;
* `vDSO` ;

that are related and very similar on system call concept.

This part will be last part in this chapter and as you can understand from the part's title - we will see what does occur in the Linux kernel when we run our programs. So, let's start.

how do we launch our programs?

There are many different ways to launch an application from an user perspective. For example we can run a program from the shell or double-click on the application icon. It does not matter. The Linux kernel handles application launch regardless how we do launch this application.

In this part we will consider the way when we just launch an application from the shell. As you know, the standard way to launch an application from shell is the following: We just launch a terminal emulator application and just write the name of the program and pass or not arguments to our program, for example:

```
~$ ls --version
ls (GNU coreutils) 8.23
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Written by Richard M. Stallman and David MacKenzie.
```

Let's consider what does occur when we launch an application from the shell, what does shell do when we write program name, what does Linux kernel do etc. But before we will start to consider these interesting things, I want to warn that this book is about the Linux kernel. That's why we will see Linux kernel insides related stuff mostly in this part. We will not consider in details what does shell do, we will not consider complex cases, for example subshells etc.

My default shell is - bash, so I will consider how do bash shell launches a program. So let's start. The `bash` shell as well as any program that written with C programming language starts from the main function. If you will look on the source code of the `bash` shell, you will find the `main` function in the shell.c source code file. This function makes many different things before the main thread loop of the `bash` started to work. For example this function:

- checks and tries to open `/dev/tty` ;
- check that shell running in debug mode;
- parses command line arguments;
- reads shell environment;
- loads `.bashrc` , `.profile` and other configuration files;
- and many many more.

After all of these operations we can see the call of the `reader_loop` function. This function defined in the eval.c source code file and represents main thread loop or in other words it reads and executes commands. As the `reader_loop` function made all checks and read the given program name and arguments, it calls the `execute_command` function from the execute_cmd.c source code file. The `execute_command` function through the chain of the functions calls:

```
execute_command
--> execute_command_internal
----> execute_simple_command
------> execute_disk_command
--------> shell_execve
```

makes different checks like do we need to start `subshell` , was it builtin `bash` function or not etc. As I already wrote above, we will not consider all details about things that are not related to the Linux kernel. In the end of this process, the `shell_execve` function calls the `execve` system call:

```
execve (command, args, env);
```

The `execve` system call has the following signature:

```
int execve(const char *filename, char *const argv [], char *const envp[]);
```

and executes a program by the given filename, with the given arguments and environment variables. This system call is the first in our case and only, for example:

```
$ strace ls
execve("/bin/ls", ["ls"], [/* 62 vars */]) = 0

$ strace echo
execve("/bin/echo", ["echo"], [/* 62 vars */]) = 0

$ strace uname
execve("/bin/uname", ["uname"], [/* 62 vars */]) = 0
```

So, an user application (`bash` in our case) calls the system call and as we already know the next step is Linux kernel.

execve system call

We saw preparation before a system call called by an user application and after a system call handler finished its work in the second part of this chapter. We stopped at the call of the `execve` system call in the previous paragraph. This system call defined in the fs/exec.c source code file and as we already know it takes three arguments:

```
SYSCALL_DEFINE3(execve,
        const char __user *, filename,
        const char __user *const __user *, argv,
        const char __user *const __user *, envp)
{
    return do_execve(getname(filename), argv, envp);
}
```

Implementation of the `execve` is pretty simple here, as we can see it just returns the result of the `do_execve` function. The `do_execve` function defined in the same source code file and do the following things:

- Initialize two pointers on a userspace data with the given arguments and environment variables;
- return the result of the `do_execveat_common` .

We can see its implementation:

```
struct user_arg_ptr argv = { .ptr.native = __argv };
struct user_arg_ptr envp = { .ptr.native = __envp };
return do_execveat_common(AT_FDCWD, filename, argv, envp, 0);
```

The `do_execveat_common` function does main work - it executes a new program. This function takes similar set of arguments, but as you can see it takes five arguments instead of three. The first argument is the file descriptor that represent directory with our application, in our case the `AT_FDCWD` means that the given pathname is interpreted relative to the current working directory of the calling process. The fifth argument is flags. In our case we passed `0` to the `do_execveat_common`. We will check in a next step, so will see it latter.

First of all the `do_execveat_common` function checks the `filename` pointer and returns if it is `NULL`. After this we check flags of the current process that limit of running processes is not exceed:

```
if (IS_ERR(filename))
    return PTR_ERR(filename);

if ((current->flags & PF_NPROC_EXCEEDED) &&
    atomic_read(&current_user()->processes) > rlimit(RLIMIT_NPROC)) {
    retval = -EAGAIN;
    goto out_ret;
}

current->flags &= ~PF_NPROC_EXCEEDED;
```

If these two checks were successful we unset `PF_NPROC_EXCEEDED` flag in the flags of the current process to prevent fail of the `execve`. You can see that in the next step we call the `unshare_files` function that defined in the kernel/fork.c and unshares the files of the current task and check the result of this function:

```
retval = unshare_files(&displaced);
if (retval)
    goto out_ret;
```

We need to call this function to eliminate potential leak of the execve'd binary's file descriptor. In the next step we start preparation of the `bprm` that represented by the `struct linux_binprm` structure (defined in the include/linux/binfmts.h header file). The `linux_binprm` structure is used to hold the arguments that are used when loading binaries. For example it contains `vma` field which has `vm_area_struct` type and represents single memory area over a contiguous interval in a given address space where our application will be loaded, `mm` field which is memory descriptor of the binary, pointer to the top of memory and many other different fields.

First of all we allocate memory for this structure with the `kzalloc` function and check the result of the allocation:

```
bprm = kzalloc(sizeof(*bprm), GFP_KERNEL);
if (!bprm)
    goto out_files;
```

After this we start to prepare the `binprm` credentials with the call of the `prepare_bprm_creds` function:

```
retval = prepare_bprm_creds(bprm);
    if (retval)
        goto out_free;

check_unsafe_exec(bprm);
current->in_execve = 1;
```

Initialization of the `binprm` credentials in other words is initialization of the `cred` structure that stored inside of the `linux_binprm` structure. The `cred` structure contains the security context of a task for example real uid of the task, real guid of the task, `uid` and `guid` for the virtual file system operations etc. In the next step as we executed preparation of the `bprm` credentials we check that now we can safely execute a program with the call of the `check_unsafe_exec` function and set the current process to the `in_execve` state.

After all of these operations we call the `do_open_execat` function that checks the flags that we passed to the `do_execveat_common` function (remember that we have `0` in the `flags`) and searches and opens executable file on disk, checks that our we will load a binary file from `noexec` mount points (we need to avoid execute a binary from filesystems that do not contain executable binaries like proc or sysfs), intializes `file` structure and returns pointer on this structure. Next we can see the call the `sched_exec` after this:

```
file = do_open_execat(fd, filename, flags);
retval = PTR_ERR(file);
if (IS_ERR(file))
    goto out_unmark;

sched_exec();
```

The `sched_exec` function is used to determine the least loaded processor that can execute the new program and to migrate the current process to it.

After this we need to check file descriptor of the give executable binary. We try to check does the name of the our binary file starts from the `/` symbol or does the path of the given executable binary is interpreted relative to the current working directory of the calling process or in other words file descriptor is `AT_FDCWD` (read above about this).

If one of these checks is successfull we set the binary parameter filename:

```
bprm->file = file;

if (fd == AT_FDCWD || filename->name[0] == '/') {
    bprm->filename = filename->name;
}
```

Otherwise if the filename is empty we set the binary parameter filename to the `/dev/fd/%d` or `/dev/fd/%d/%s` depends on the filename of the given executable binary which means that we will execute the file to which the file descriptor refers:

```
} else {
    if (filename->name[0] == '\0')
        pathbuf = kasprintf(GFP_TEMPORARY, "/dev/fd/%d", fd);
    else
        pathbuf = kasprintf(GFP_TEMPORARY, "/dev/fd/%d/%s",
                            fd, filename->name);
    if (!pathbuf) {
        retval = -ENOMEM;
        goto out_unmark;
    }

    bprm->filename = pathbuf;
}

bprm->interp = bprm->filename;
```

Note that we set not only the `bprm->filename` but also `bprm->interp` that will contain name of the program interpreter. For now we just write the same name there, but later it will be updated with the real name of the program interpreter depends on binary format of a program. You can read above that we already prepared `cred` for the `linux_binprm`. The next step is initalization of other fields of the `linux_binprm`. First of all we call the `bprm_mm_init` function and pass the `bprm` to it:

```
retval = bprm_mm_init(bprm);
if (retval)
    goto out_unmark;
```

The `bprm_mm_init` defined in the same source code file and as we can understand from the function's name, it makes initialization of the memory descriptor or in other words the `bprm_mm_init` function initializes `mm_struct` structure. This structure defined in the include/linux/mm_types.h header file and represents address space of a process. We will not consider implementation of the `bprm_mm_init` function because we do not know many important stuff related to the Linux kernel memory manager, but we just need to know that this function initializes `mm_struct` and populate it with a temporary stack `vm_area_struct`.

After this we calculate the count of the command line arguments which are were passed to the our executable binary, the count of the environment variables and set it to the `bprm->argc` and `bprm->envc` respectively:

```
bprm->argc = count(argv, MAX_ARG_STRINGS);
if ((retval = bprm->argc) < 0)
    goto out;

bprm->envc = count(envp, MAX_ARG_STRINGS);
if ((retval = bprm->envc) < 0)
    goto out;
```

As you can see we do this operations with the help of the `count` function that defined in the same source code file and calculates the count of strings in the `argv` array. The `MAX_ARG_STRINGS` macro defined in the include/uapi/linux/binfmts.h header file and as we can understand from the macro's name, it represents maximum number of strings that were passed to the `execve` system call. The value of the `MAX_ARG_STRINGS` :

```
#define MAX_ARG_STRINGS 0x7FFFFFFF
```

After we calculated the number of the command line arguments and environment variables, we call the `prepare_binprm` function. We already call the function with the similar name before this moment. This function is called `prepare_binprm_cred` and we remember that this function initializes `cred` structure in the `linux_bprm` . Now the `prepare_binprm` function:

```
retval = prepare_binprm(bprm);
if (retval < 0)
    goto out;
```

fills the `linux_binprm` structure with the `uid` from inode and read `128` bytes from the binary executable file. We read only first `128` from the executable file because we need to check a type of our executable. We will read the rest of the executable file in the later step. After the preparation of the `linux_bprm` structure we copy the filename of the executable binary file, command line arguments and enviroment variables to the `linux_bprm` with the call of the `copy_strings_kernel` function:

```
retval = copy_strings_kernel(1, &bprm->filename, bprm);
if (retval < 0)
    goto out;

retval = copy_strings(bprm->envc, envp, bprm);
if (retval < 0)
    goto out;

retval = copy_strings(bprm->argc, argv, bprm);
if (retval < 0)
    goto out;
```

And set the pointer to the top of new program's stack that we set in the `bprm_mm_init` function:

```
bprm->exec = bprm->p;
```

The top of the stack will contain the program filename and we store this fileneme tothe `exec` field of the `linux_bprm` structure.

Now we have filled `linux_bprm` structure, we call the `exec_binprm` function:

```
retval = exec_binprm(bprm);
if (retval < 0)
    goto out;
```

First of all we store the pid and `pid` that seen from the namespace of the current task in the `exec_binprm` :

```
old_pid = current->pid;
rcu_read_lock();
old_vpid = task_pid_nr_ns(current, task_active_pid_ns(current->parent));
rcu_read_unlock();
```

and call the:

```
search_binary_handler(bprm);
```

function. This function goes through the list of handlers that contains different binary formats. Currently the Linux kernel supports following binary formats:

- `binfmt_script` - support for interpreted scripts that are starts from the #! line;
- `binfmt_misc` - support differnt binary formats, according to runtime configuration of the Linux kernel;

* `binfmt_elf` - support elf format;
* `binfmt_aout` - support a.out format;
* `binfmt_flat` - support for flat format;
* `binfmt_elf_fdpic` - Support for elf FDPIC binaries;
* `binfmt_em86` - support for Intel elf binaries running on Alpha machines.

So, the search-binary_handler tries to call the `load_binary` function and pass `linux_binprm` to it. If the binary handler supports the given executable file format, it starts to prepare the executable binary for execution:

```
int search_binary_handler(struct linux_binprm *bprm)
{
    ...
    ...
    ...
    list_for_each_entry(fmt, &formats, lh) {
        retval = fmt->load_binary(bprm);
        if (retval < 0 && !bprm->mm) {
            force_sigsegv(SIGSEGV, current);
            return retval;
        }
    }

    return retval;
```

Where the `load_binary` for example for the elf checks the magic number (each `elf` binary file contains magic number in the header) in the `linux_bprm` buffer (remember that we read first `128` bytes from the executable binary file): and exit if it is not `elf` binary:

```
static int load_elf_binary(struct linux_binprm *bprm)
{
    ...
    ...
    ...
    loc->elf_ex = *((struct elfhdr *)bprm->buf);

    if (memcmp(elf_ex.e_ident, ELFMAG, SELFMAG) != 0)
        goto out;
```

If the given executable file is in `elf` format, the `load_elf_binary` continues to execute. The `load_elf_binary` does many different things to prepare on execution executable file. For example it checks the architecture and type of the executable file:

```
if (loc->elf_ex.e_type != ET_EXEC && loc->elf_ex.e_type != ET_DYN)
    goto out;
if (!elf_check_arch(&loc->elf_ex))
    goto out;
```

and exit if there is wrong architecture and executable file non executable non shared. Tries to load the `program header table` :

```
elf_phdata = load_elf_phdrs(&loc->elf_ex, bprm->file);
if (!elf_phdata)
    goto out;
```

that describes segments. Read the `program interpreter` and libraries that linked with the our executable binary file from disk and load it to memory. The `program interpreter` specified in the `.interp` section of the executable file and as you can read in the part that describes Linkers it is - `/lib64/ld-linux-x86-64.so.2` for the `x86_64` . It setups the stack and map `elf` binary into the correct location in memory. It maps the bss and the brk sections and does many many other different things to prepare executable file to execute.

In the end of the execution of the `load_elf_binary` we call the `start_thread` function and pass three arguments to it:

```
    start_thread(regs, elf_entry, bprm->p);
    retval = 0;
out:
    kfree(loc);
out_ret:
    return retval;
```

These arguments are:

- Set of registers for the new task;
- Address of the entry point of the new task;
- Address of the top of the stack for the new task.

As we can understand from the function's name, it starts new thread, but it is not so. The `start_thread` function just prepares new task's registers to be ready to run. Let's look on the implementation of this function:

```
void
start_thread(struct pt_regs *regs, unsigned long new_ip, unsigned long new_sp)
{
        start_thread_common(regs, new_ip, new_sp,
                            __USER_CS, __USER_DS, 0);
}
```

As we can see the `start_thread` function just makes a call of the `start_thread_common` function that will do all for us:

```
static void
start_thread_common(struct pt_regs *regs, unsigned long new_ip,
                    unsigned long new_sp,
                    unsigned int _cs, unsigned int _ss, unsigned int _ds)
{
        loadsegment(fs, 0);
        loadsegment(es, _ds);
        loadsegment(ds, _ds);
        load_gs_index(0);
        regs->ip                = new_ip;
        regs->sp                = new_sp;
        regs->cs                = _cs;
        regs->ss                = _ss;
        regs->flags             = X86_EFLAGS_IF;
        force_iret();
}
```

The `start_thread_common` function fills `fs` segment register with zero and `es` and `ds` with the value of the data segment register. After this we set new values to the instruction pointer, `cs` segments etc. In the end of the `start_thread_common` function we can see the `force_iret` macro that force a system call return via `iret` instruction. Ok, we prepared new thread to run in userspace and now we can return from the `exec_binprm` and now we are in the `do_execveat_common` again. After the `exec_binprm` will finish its execution we release memory for structures that was allocated before and return.

After we returned from the `execve` system call handler, execution of our program will be started. We can do it, because all context related information already configured for this purpose. As we saw the `execve` system call does not return control to a process, but code, data and other segments of the caller process are just overwritten of the program segments. The exit from our application will be implemented through the `exit` system call.

That's all. From this point our programm will be executed.

Conclusion

This is the end of the fourth and last part of the about the system calls concept in the Linux kernel. We saw almost all related stuff to the `system call` concept in these four parts. We started from the understanding of the `system call` concept, we have learned what is it and why do users applications need in this concept. Next we saw how does the Linux handle a system call from an user application. We met two similar concepts to the `system call` concept, they are `vsyscall` and `vDSO` and finally we saw how does Linux kernel run an user program.

If you have questions or suggestions, feel free to ping me in twitter 0xAX, drop me email or just create issue.

Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-insides.

Links

- System call
- shell
- bash
- entry point
- C
- environment variables
- file descriptor
- real uid
- virtual file system
- procfs
- sysfs
- inode
- pid
- namespace
- #!
- elf
- a.out
- flat
- Alpha
- FDPIC
- segments
- Linkers
- Processor register
- instruction pointer

* Previous part

How does the `open` system call work

Introduction

This is the fifth part of the chapter that describes system calls mechanism in the Linux kernel. Previous parts of this chapter described this mechanism in general. Now I will try to describe implementation of different system calls in the Linux kernel. Previous parts from this chapter and parts from other chapters of the books describe mostly deep parts of the Linux kernel that are faintly visible or fully invisible from the userspace. But the Linux kernel code is not only about itself. The vast of the Linux kernel code provides ability to our code. Due to the linux kernel our programs can read/write from/to files and don't know anything about sectors, tracks and other parts of a disk structures, we can send data over network and don't build encapsulated network packets by hand and etc.

I don't know how about you, but it is interesting to me not only how an operating system works, but how do my software interacts with it. As you may know, our programs interacts with the kernel through the special mechanism which is called system call. So, I've decided to write series of parts which will describe implementation and behavior of system calls which we are using every day like `read` , `write` , `open` , `close` , `dup` and etc.

I have decided to start from the description of the open system call. if you have written at least one `C` program, you should know that before we are able to read/write or execute other manipulations with a file we need to open it with the `open` function:

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

int main(int argc, char *argv) {
        int fd = open("test", O_RDONLY);

        if fd < 0 {
                perror("Opening of the file is failed\n");
        }
        else {
                printf("file sucessfully opened\n");
        }

        close(fd);
        return 0;
}
```

In this case, the open is the function from standard library, but not system call. The standard library will call related system call for us. The `open` call will return a file descriptor which is just a unique number within our process which is associated with the opened file. Now as we opened a file and got file descriptor as result of `open` call, we may start to interact with this file. We can write into, read from it and etc. List of opened file by a process is available via proc filesystem:

```
$ sudo ls /proc/1/fd/

0   10  12  14  16  2   21  23  25  27  29  30  32  34  36  38  4   41  43  45  47  49
 50  53  55  58  6   61  63  67  8
1   11  13  15  19  20  22  24  26  28  3   31  33  35  37  39  40  42  44  46  48  5
 51  54  57  59  60  62  65  7   9
```

I am not going to describe more details about the `open` routine from the userspace view in this post, but mostly from the kernel side. if you are not very familiar with, you can get more info in the man page.

So let's start.

Definition of the open system call

If you have read the fourth part of the linux-insides book, you should know that system calls are defined with the help of `SYSCALL_DEFINE` macro. So, the `open` system call is not exception.

Definition of the `open` system call is located in the fs/open.c source code file and looks pretty small for the first view:

```
SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode)
{
    if (force_o_largefile())
        flags |= O_LARGEFILE;

    return do_sys_open(AT_FDCWD, filename, flags, mode);
}
```

As you may guess, the `do_sys_open` function from the same source code file does the main job. But before this function will be called, let's consider the `if` clause from which the implementation of the `open` system call starts:

```
if (force_o_largefile())
    flags |= O_LARGEFILE;
```

Here we apply the `O_LARGEFILE` flag to the flags which were passed to `open` system call in a case when the `force_o_largefile()` will return true. What is `O_LARGEFILE`? We may read this in the man page for the `open(2)` system call:

> O_LARGEFILE
>
> (LFS) Allow files whose sizes cannot be represented in an off_t (but can be represented in an off64_t) to be opened.

As we may read in the GNU C Library Reference Manual:

> off_t
>
> This is a signed integer type used to represent file sizes. In the GNU C Library, this type is no narrower than int. If the source is compiled with _FILE_OFFSET_BITS == 64 this type is transparently replaced by off64_t.

and

> off64_t
>
> This type is used similar to off_t. The difference is that even on 32 bit machines, where the off_t type would have 32 bits, off64_t has 64 bits and so is able to address files up to 2^63 bytes in length. When compiling with _FILE_OFFSET_BITS == 64 this type is available under the name off_t.

So it is not hard to guess that the `off_t`, `off64_t` and `O_LARGEFILE` are about a file size. In the case of the Linux kernel, the `O_LARGEFILE` is used to disallow opening large files on 32bit systems if the caller didn't specify `O_LARGEFILE` flag during opening of a file. On 64bit systems we force on this flag in open system call. And the `force_o_largefile` macro from the include/linux/fcntl.h linux kernel header file confirms this:

```
#ifndef force_o_largefile
#define force_o_largefile() (BITS_PER_LONG != 32)
#endif
```

This macro may be architecture-specific as for example for IA-64 architecture, but in our case the x86_64 does not provide definition of the `force_o_largefile` and it will be used from include/linux/fcntl.h.

So, as we may see the `force_o_largefile` is just a macro which expands to the `true` value in our case of x86_64 architecture. As we are considering 64-bit architecture, the `force_o_largefile` will be expanded to `true` and the `O_LARGEFILE` flag will be added to the set of flags which were passed to the `open` system call.

Now as we considered meaning of the `O_LARGEFILE` flag and `force_o_largefile` macro, we can proceed to the consideration of the implementation of the `do_sys_open` function. As I wrote above, this function is defined in the same source code file and looks:

```
long do_sys_open(int dfd, const char __user *filename, int flags, umode_t mode)
{
    struct open_flags op;
    int fd = build_open_flags(flags, mode, &op);
    struct filename *tmp;

    if (fd)
        return fd;

    tmp = getname(filename);
    if (IS_ERR(tmp))
        return PTR_ERR(tmp);

    fd = get_unused_fd_flags(flags);
    if (fd >= 0) {
        struct file *f = do_filp_open(dfd, tmp, &op);
        if (IS_ERR(f)) {
            put_unused_fd(fd);
            fd = PTR_ERR(f);
        } else {
            fsnotify_open(f);
            fd_install(fd, f);
        }
    }
    putname(tmp);
    return fd;
}
```

Let's try to understand how the `do_sys_open` works step by step.

open(2) flags

As you know the `open` system call takes set of `flags` as second argument that control opening a file and `mode` as third argument that specifies permission the permissions of a file if it is created. The `do_sys_open` function starts from the call of the `build_open_flags` function which does some checks that set of the given flags is valid and handles different conditions of flags and mode.

Let's look at the implementation of the `build_open_flags`. This function is defined in the same kernel file and takes three arguments:

- flags - flags that control opening of a file;
- mode - permissions for newly created file;

The last argument - `op` is represented with the `open_flags` structure:

```
struct open_flags {
        int open_flag;
        umode_t mode;
        int acc_mode;
        int intent;
        int lookup_flags;
};
```

which is defined in the fs/internal.h header file and as we may see it holds information about flags and access mode for internal kernel purposes. As you already may guess the main goal of the `build_open_flags` function is to fill an instance of this structure.

Implementation of the `build_open_flags` function starts from the definition of local variables and one of them is:

```
int acc_mode = ACC_MODE(flags);
```

This local variable represents access mode and its initial value will be equal to the value of expanded `ACC_MODE` macro. This macro is defined in the include/linux/fs.h and looks pretty interesting:

```
#define ACC_MODE(x) ("\004\002\006\006"[(x)&O_ACCMODE])
#define O_ACCMODE   00000003
```

The `"\004\002\006\006"` is an array of four chars:

```
"\004\002\006\006" == {'\004', '\002', '\006', '\006'}
```

So, the `ACC_MODE` macro just expands to the accession to this array by `[(x) & O_ACCMODE]` index. As we just saw, the `O_ACCMODE` is `00000003`. By applying `x & O_ACCMODE` we will take the two least significant bits which are represents `read`, `write` or `read/write` access modes:

```
#define O_RDONLY        00000000
#define O_WRONLY        00000001
#define O_RDWR          00000002
```

After getting value from the array by the calculated index, the `ACC_MODE` will be expanded to access mode mask of a file which will hold `MAY_WRITE`, `MAY_READ` and other information.

We may see following condition after we have calculated initial access mode:

```
if (flags & (O_CREAT | __O_TMPFILE))
    op->mode = (mode & S_IALLUGO) | S_IFREG;
else
    op->mode = 0;
```

Here we reset permissions in `open_flags` instance if a opened file wasn't temporary and wasn't open for creation. This is because:

> if neither O_CREAT nor O_TMPFILE is specified, then mode is ignored.

In other case if `O_CREAT` or `O_TMPFILE` were passed we canonicalize it to a regular file because a directory should be created with the opendir system call.

At the next step we check that a file is not tried to be opened via fanotify and without the `O_CLOEXEC` flag:

```
flags &= ~FMODE_NONOTIFY & ~O_CLOEXEC;
```

We do this to not leak a file descriptor. By default, the new file descriptor is set to remain open across an `execve` system call, but the `open` system call supports `O_CLOEXEC` flag that can be used to change this default behaviour. So we do this to prevent leaking of a file descriptor when one thread opens a file to set `O_CLOEXEC` flag and in the same time the second process does a fork) + execve) and as you may remember that child will have copies of the parent's set of open file descriptors.

At the next step we check that if our flags contains `O_SYNC` flag, we apply `O_DSYNC` flag too:

```
if (flags & __O_SYNC)
    flags |= O_DSYNC;
```

The `O_SYNC` flag guarantees that the any write call will not return before all data has been transferred to the disk. The `O_DSYNC` is like `O_SYNC` except that there is no requirement to wait for any metadata (like `atime`, `mtime` and etc.) changes will be written. We apply `O_DSYNC` in a case of `__O_SYNC` because it is implemented as `__O_SYNC|O_DSYNC` in the Linux kernel.

After this we must be sure that if a user wants to create temporary file, the flags should contain `O_TMPFILE_MASK` or in other words it should contain or `O_CREAT` or `O_TMPFILE` or both and also it should be writeable:

```
if (flags & __O_TMPFILE) {
    if ((flags & O_TMPFILE_MASK) != O_TMPFILE)
        return -EINVAL;
    if (!(acc_mode & MAY_WRITE))
        return -EINVAL;
} else if (flags & O_PATH) {
           flags &= O_DIRECTORY | O_NOFOLLOW | O_PATH;
        acc_mode = 0;
}
```

as it is written in in the manual page:

> O_TMPFILE must be specified with one of O_RDWR or O_WRONLY

If we didn't pass `O_TMPFILE` for creation of a temporary file, we check the `O_PATH` flag at the next condition. The `O_PATH` flag allows us to obtain a file descriptor that may be used for two following purposes:

- to indicate a location in the filesystem tree;
- to perform operations that act purely at the file descriptor level.

So, in this case the file itself is not opened, but operations like `dup`, `fcntl` and other can be used. So, if all file content related operations like `read`, `write` and other are permitted, only `O_DIRECTORY | O_NOFOLLOW | O_PATH` flags can be used. We have finished with flags for this moment in the `build_open_flags` for this moment and we may fill our `open_flags->open_flag` with them:

```
op->open_flag = flags;
```

Now we have filled `open_flag` field which represents flags that will control opening of a file and `mode` that will represent `umask` of a new file if we open file for creation. There are still to fill last flags in the our `open_flags` structure. The next is `op->acc_mode` which represents access mode to a opened file. We already filled the `acc_mode` local variable with the initial value at the beginning of the `build_open_flags` and now we check last two flags related to access mode:

```
if (flags & O_TRUNC)
        acc_mode |= MAY_WRITE;
if (flags & O_APPEND)
    acc_mode |= MAY_APPEND;
op->acc_mode = acc_mode;
```

These flags are - `O_TRUNC` that will truncate an opened file to length `0` if it existed before we open it and the `O_APPEND` flag allows to open a file in `append mode`. So the opened file will be appended during write but not overwritten.

The next field of the `open_flags` structure is - `intent`. It allows us to know about our intention or in other words what do we really want to do with file, open it, create, rename it or something else. So we set it to zero if our flags contains the `O_PATH` flag as we can't do anything related to a file content with this flag:

```
op->intent = flags & O_PATH ? 0 : LOOKUP_OPEN;
```

or just to `LOOKUP_OPEN` intention. Additionally we set `LOOKUP_CREATE` intention if we want to create new file and to be sure that a file didn't exist before with `O_EXCL` flag:

```
if (flags & O_CREAT) {
    op->intent |= LOOKUP_CREATE;
    if (flags & O_EXCL)
        op->intent |= LOOKUP_EXCL;
}
```

The last flag of the `open_flags` structure is the `lookup_flags`:

```
if (flags & O_DIRECTORY)
    lookup_flags |= LOOKUP_DIRECTORY;
if (!(flags & O_NOFOLLOW))
    lookup_flags |= LOOKUP_FOLLOW;
op->lookup_flags = lookup_flags;

return 0;
```

We fill it with `LOOKUP_DIRECTORY` if we want to open a directory and `LOOKUP_FOLLOW` if we don't want to follow (open) symlink. That's all. It is the end of the `build_open_flags` function. The `open_flags` structure is filled with modes and flags for a file opening and we can return back to the `do_sys_open`.

Actual opening of a file

At the next step after `build_open_flags` function is finished and we have formed flags and modes for our file we should get the `filename` structure with the help of the `getname` function by name of a file which was passed to the `open` system call:

```
tmp = getname(filename);
if (IS_ERR(tmp))
    return PTR_ERR(tmp);
```

The `getname` function is defined in the fs/namei.c source code file and looks:

```
struct filename *
getname(const char __user * filename)
{
        return getname_flags(filename, 0, NULL);
}
```

So, it just calls the `getname_flags` function and returns its result. The main goal of the `getname_flags` function is to copy a file path given from userland to kernel space. The `filename` structure is defined in the include/linux/fs.h linux kernel header file and contains following fields:

- name - pointer to a file path in kernel space;
- uptr - original pointer from userland;
- aname - filename from audit context;
- refcnt - reference counter;
- iname - a filename in a case when it will be less than `PATH_MAX` .

As I already wrote above, the main goal of the `getname_flags` function is to copy name of a file which was passed to the `open` system call from user space to kernel space with the strncpy_from_user function. The next step after a filename will be copied to kernel space is getting of new non-busy file descriptor:

```
fd = get_unused_fd_flags(flags);
```

The `get_unused_fd_flags` function takes table of open files of the current process, minimum (`0`) and maximum (`RLIMIT_NOFILE`) possible number of a file descriptor in the system and flags that we have passed to the `open` system call and allocates file descriptor and mark it busy in the file descriptor table of the current process. The `get_unused_fd_flags` function sets or clears the `O_CLOEXEC` flag depends on its state in the passed flags.

The last and main step in the `do_sys_open` is the `do_filp_open` function:

```C
struct file *f = do_filp_open(dfd, tmp, &op);

if (IS_ERR(f)) {
    put_unused_fd(fd);
    fd = PTR_ERR(f);
} else {
    fsnotify_open(f);
    fd_install(fd, f);
}
```

The main goal of this function is to resolve given path name into `file` structure which represents an opened file of a process. If something going wrong and execution of the `do_filp_open` function will be failed, we should free new file descriptor with the `put_unused_fd` or in other way the `file` structure returned by the `do_filp_open` will be stored in the file descriptor table of the current process.

Now let's take a short look at the implementation of the `do_filp_open` function. This function is defined in the fs/namei.c linux kernel source code file and starts from initialization of the `nameidata` structure. This structure will provide a link to a file inode. Actually this is one of the main point of the `do_filp_open` function to acquire an `inode` by the filename given to `open` system call. After the `nameidata` structure will be initialized, the `path_openat` function will be called:

```C
filp = path_openat(&nd, op, flags | LOOKUP_RCU);

if (unlikely(filp == ERR_PTR(-ECHILD)))
    filp = path_openat(&nd, op, flags);
if (unlikely(filp == ERR_PTR(-ESTALE)))
    filp = path_openat(&nd, op, flags | LOOKUP_REVAL);
```

Note that it is called three times. Actually, the Linux kernel will open the file in RCU mode. This is the most efficient way to open a file. If this try will be failed, the kernel enters the normal mode. The third call is relatively rare, only in the nfs file system is likely to be used. The `path_openat` function executes `path lookup` or in other words it tries to find a `dentry` (what the Linux kernel uses to keep track of the hierarchy of files in directories) corresponding to a path.

The `path_openat` function starts from the call of the `get_empty_flip()` function that allocates a new `file` structure with some additional checks like do we exceed amount of opened files in the system or not and etc. After we have got allocated new `file` structure we call the `do_tmpfile` or `do_o_path` functions in a case if we have passed `O_TMPFILE |`

`O_CREATE` or `O_PATH` flags during call of the `open` system call. These both cases are quite specific, so let's consider quite usual case when we want to open already existed file and want to read/write from/to it.

In this case the `path_init` function will be called. This function performs some preporatory work before actual path lookup. This includes search of start position of path traversal and its metadata like `inode` of the path, `dentry inode` and etc. This can be `root` directory - `/` or current directory as in our case, because we use `AT_CWD` as starting point (see call of the `do_sys_open` at the beginning of the post).

The next step after the `path_init` is the loop which executes the `link_path_walk` and `do_last`. The first function executes name resolution or in other words this function starts process of walking along a given path. It handles everything step by step except the last component of a file path. This handling includes checking of a permissions and getting a file component. As a file component is gotten, it is passed to `walk_component` that updates current directory entry from the `dcache` or asks underlying filesystem. This repeats before all path's components will not be handled in such way. After the `link_path_walk` will be executed, the `do_last` function will populate a `file` structure based on the result of the `link_path_walk`. As we reached last component of the given file path the `vfs_open` function from the `do_last` will be called.

This function is defined in the fs/open.c linux kernel source code file and the main goal of this function is to call an `open` operation of underlying filesystem.

That's all for now. We didn't consider **full** implementation of the `open` system call. We skip some parts like handling case when we want to open a file from other filesystem with different mount point, resolving symlinks and etc., but it should be not so hard to follow this stuff. This stuff does not included in **generic** implementation of open system call and depends on underlying filesystem. If you are interested in, you may lookup the `file_operations.open` callback function for a certain filesystem.

Conclusion

This is the end of the fifth part of the implementation of different system calls in the Linux kernel. If you have questions or suggestions, ping me on twitter 0xAX, drop me an email, or just create an issue. In the next part, we will continue to dive into system calls in the Linux kernel and see the implementation of the read system call.

Please note that English is not my first language and I am really sorry for any inconvenience. If you find any mistakes please send me PR to linux-insides.

Links

- system call
- open
- file descriptor
- proc
- GNU C Library Reference Manual
- IA-64
- x86_64
- opendir
- fanotify
- fork)
- execve)
- symlink
- audit
- inode
- RCU
- read
- previous part

定时器和时钟管理

本章介绍 Linux 内核中定时器和时钟管理相关的观念。

- 简介 - 简单介绍 Linux 内核中的定时器。
- 时钟源框架简介 - this part describes `clocksource` framework in the Linux kernel.
- The tick broadcast framework and dyntick - 介绍 tick broadcast framework and dyntick 概念。
- 定时器介绍 - 介绍 Linux 内核中的定时器。
- Clockevents 框架简介 - 介绍另外一个时钟管理相关的框架：`clockevents`.
- x86 相关的时钟源 - 介绍 `x86_64` 相关的时钟源。
- Linux 内核中与时钟相关的系统调用 - 介绍时钟相关的系统调用。

Timers and time management in the Linux kernel. Part 1.

Introduction

This is yet another post that opens new chapter in the linux-insides book. The previous part was a list part of the chapter that describes system call concept and now time is to start new chapter. As you can understand from the post's title, this chapter will be devoted to the `timers` and `time management` in the Linux kernel. The choice of topic for the current chapter is not accidental. Timers and generally time management are very important and widely used in the Linux kernel. The Linux kernel uses timers for various tasks, different timeouts for example in TCP implementation, the kernel must know current time, scheduling asynchronous functions, next event interrupt scheduling and many many more.

So, we will start to learn implementation of the different time management related stuff in this part. We will see different types of timers and how do different Linux kernel subsystems use them. As always we will start from the earliest part of the Linux kernel and will go through initialization process of the Linux kernel. We already did it in the special chapter which describes initialization process of the Linux kernel, but as you may remember we missed some things there. And one of them is the initialization of timers.

Let's start.

Initialization of non-standard PC hardware clock

After the Linux kernel was decompressed (more about this you can read in the Kernel decompression part) the architecture non-specific code starts to work in the init/main.c source code file. After initialization of the lock validator, initialization of cgroups and setting canary value we can see the call of the `setup_arch` function.

As you may remember this function defined in the arch/x86/kernel/setup.c source code file and prepares/initializes architecture-specific stuff (for example it reserves place for bss section, reserves place for initrd, parses kernel command line and many many other things). Besides this, we can find some time management related functions there.

The first is:

```
x86_init.timers.wallclock_init();
```

We already saw `x86_init` structure in the chapter that describes initialization of the Linux kernel. This structure contains pointers to the default setup functions for the different platforms like Intel MID, Intel CE4100 and etc. The `x86_init` structure defined in the arch/x86/kernel/x86_init.c and as you can see it determines standard PC hardware by default.

As we can see, the `x86_init` structure has `x86_init_ops` type that provides a set of functions for platform specific setup like reserving standard resources, platform specific memory setup, initialization of interrupt handlers and etc. This structure looks like:

```
struct x86_init_ops {
    struct x86_init_resources       resources;
    struct x86_init_mpparse         mpparse;
    struct x86_init_irqs            irqs;
    struct x86_init_oem             oem;
    struct x86_init_paging          paging;
    struct x86_init_timers          timers;
    struct x86_init_iommu           iommu;
    struct x86_init_pci             pci;
};
```

We can note `timers` field that has `x86_init_timers` type and as we can understand by its name - this field is related to time management and timers. The `x86_init_timers` contains four fields which are all functions that returns pointer on void:

- `setup_percpu_clockev` - set up the per cpu clock event device for the boot cpu;
- `tsc_pre_init` - platform function called before TSC init;
- `timer_init` - initialize the platform timer;
- `wallclock_init` - initialize the wallclock device.

So, as we already know, in our case the `wallclock_init` executes initialization of the wallclock device. If we will look on the `x86_init` structure, we will see that `wallclock_init` points to the `x86_init_noop` :

```
struct x86_init_ops x86_init __initdata = {
    ...
    ...
    ...
    .timers = {
        .wallclock_init            = x86_init_noop,
    },
    ...
    ...
    ...
}
```

Where the `x86_init_noop` is just a function that does nothing:

```
void __cpuinit x86_init_noop(void) { }
```

for the standard PC hardware. Actually, the `wallclock_init` function is used in the Intel MID platform. Initialization of the `x86_init.timers.wallclock_init` located in the arch/x86/platform/intel-mid/intel-mid.c source code file in the `x86_intel_mid_early_setup` function:

```
void __init x86_intel_mid_early_setup(void)
{
    ...
    ...
    ...
    x86_init.timers.wallclock_init = intel_mid_rtc_init;
    ...
    ...
    ...
}
```

Implementation of the `intel_mid_rtc_init` function is in the arch/x86/platform/intel-mid/intel_mid_vrtc.c source code file and looks pretty easy. First of all, this function parses Simple Firmware Interface M-Real-Time-Clock table for the getting such devices to the `sfi_mrtc_array` array and initialization of the `set_time` and `get_time` functions:

```
void __init intel_mid_rtc_init(void)
{
    unsigned long vrtc_paddr;

    sfi_table_parse(SFI_SIG_MRTC, NULL, NULL, sfi_parse_mrtc);

    vrtc_paddr = sfi_mrtc_array[0].phys_addr;
    if (!sfi_mrtc_num || !vrtc_paddr)
        return;

    vrtc_virt_base = (void __iomem *)set_fixmap_offset_nocache(FIX_LNW_VRTC,
                                vrtc_paddr);

    x86_platform.get_wallclock = vrtc_get_time;
    x86_platform.set_wallclock = vrtc_set_mmss;
}
```

That's all, after this a device based on `Intel MID` will be able to get time from hardware clock. As I already wrote, the standard PC x86_64 architecture does not support `x86_init_noop` and just do nothing during call of this function. We just saw initialization of the real time clock for the Intel MID architecture and now times to return to the general `x86_64` architecture and will look on the time management related stuff there.

Acquainted with jiffies

If we will return to the `setup_arch` function which is located as you remember in the arch/x86/kernel/setup.c source code file, we will see the next call of the time management related function:

```
register_refined_jiffies(CLOCK_TICK_RATE);
```

Before we will look on the implementation of this function, we must know about jiffy. As we can read on wikipedia:

```
Jiffy is an informal term for any unspecified short period of time
```

This definition is very similar to the `jiffy` in the Linux kernel. There is global variable with the `jiffies` which holds the number of ticks that have occurred since the system booted. The Linux kernel sets this variable to zero:

```
extern unsigned long volatile __jiffy_data jiffies;
```

during initialization process. This global variable will be increased each time during timer interrupt. Besides this, near the `jiffies` variable we can see definition of the similar variable

```
extern u64 jiffies_64;
```

Actually only one of these variables is in use in the Linux kernel. And it depends on the processor type. For the x86_64 it will be `u64` use and for the x86 is `unsigned long`. We will see this if we will look on the arch/x86/kernel/vmlinux.lds.S linker script:

```
#ifdef CONFIG_X86_32
...
jiffies = jiffies_64;
...
#else
...
jiffies_64 = jiffies;
...
#endif
```

In the case of `x86_32` the `jiffies` will be lower `32` bits of the `jiffies_64` variable. Schematically, we can imagine it as follows

```
                    jiffies_64
+-----------------------------------------------------+
|                       |                             |
|                       |                             |
|                       |       jiffies on `x86_32`   |
|                       |                             |
|                       |                             |
+-----------------------------------------------------+
63                     31                             0
```

Now we know a little theory about `jiffies` and we can return to the our function. There is no architecture-specific implementation for our function - the `register_refined_jiffies`. This function located in the generic kernel code - kernel/time/jiffies.c source code file. Main point of the `register_refined_jiffies` is registration of the jiffy `clocksource`. Before we will look on the implementation of the `register_refined_jiffies` function, we must know what is it `clocksource`. As we can read in the comments:

```
The `clocksource` is hardware abstraction for a free-running counter.
```

I'm not sure about you, but that description didn't give a good understanding about the `clocksource` concept. Let's try to understand what is it, but we will not go deeper because this topic will be described in a separate part in much more detail. The main point of the `clocksource` is timekeeping abstraction or in very simple words - it provides a time value to the kernel. We already know about `jiffies` interface that represents number of ticks that have occurred since the system booted. It represented by the global variable in the Linux kernel and increased each timer interrupt. The Linux kernel can use `jiffies` for time measurement. So why do we need in separate context like the `clocksource` ? Actually different hardware devices provide different clock sources that are widely in their capabilities. The availability of more precise techniques for time intervals measurement is hardware-dependent.

For example `x86` has on-chip a 64-bit counter that is called Time Stamp Counter and its frequency can be equal to processor frequency. Or for example High Precision Event Timer that consists of a `64-bit` counter of at least `10 MHz` frequency. Two different timers and they are both for `x86` . If we will add timers from other architectures, this only makes this problem more complex. The Linux kernel provides `clocksource` concept to solve the problem.

The clocksource concept represented by the `clocksource` structure in the Linux kernel. This structure defined in the include/linux/clocksource.h header file and contains a couple of fields that describe a time counter. For example it contains - `name` field which is the name of a counter, `flags` field that describes different properties of a counter, pointers to the `suspend` and `resume` functions, and many more.

Let's look on the `clocksource` structure for jiffies that defined in the kernel/time/jiffies.c source code file:

```
static struct clocksource clocksource_jiffies = {
    .name         = "jiffies",
    .rating         = 1,
    .read         = jiffies_read,
    .mask         = 0xffffffff,
    .mult         = NSEC_PER_JIFFY << JIFFIES_SHIFT,
    .shift         = JIFFIES_SHIFT,
    .max_cycles     = 10,
};
```

We can see definition of the default name here - `jiffies` , the next is `rating` field allows the best registered clock source to be chosen by the clock source management code available for the specified hardware. The `rating` may have following value:

- `1-99` - Only available for bootup and testing purposes;
- `100-199` - Functional for real use, but not desired.

- `200-299` - A correct and usable clocksource.
- `300-399` - A reasonably fast and accurate clocksource.
- `400-499` - The ideal clocksource. A must-use where available;

For example rating of the time stamp counter is `300`, but rating of the high precision event timer is `250`. The next field is `read` - is pointer to the function that allows to read clocksource's cycle value or in other words it just returns `jiffies` variable with `cycle_t` type:

```
static cycle_t jiffies_read(struct clocksource *cs)
{
        return (cycle_t) jiffies;
}
```

that is just 64-bit unsigned type:

```
typedef u64 cycle_t;
```

The next field is the `mask` value ensures that subtraction between counters values from non `64 bit` counters do not need special overflow logic. In our case the mask is `0xffffffff` and it is `32` bits. This means that `jiffy` wraps around to zero after `42` seconds:

```
>>> 0xffffffff
4294967295
# 42 nanoseconds
>>> 42 * pow(10, -9)
4.2000000000000006e-08
# 43 nanoseconds
>>> 43 * pow(10, -9)
4.3e-08
```

The next two fields `mult` and `shift` are used to convert the clocksource's period to nanoseconds per cycle. When the kernel calls the `clocksource.read` function, this function returns value in `machine` time units represented with `cycle_t` data type that we saw just now. To convert this return value to the nanoseconds we need in these two fields: `mult` and `shift`. The `clocksource` provides `clocksource_cyc2ns` function that will do it for us with the following expression:

```
((u64) cycles * mult) >> shift;
```

As we can see the `mult` field is equal:

```
NSEC_PER_JIFFY << JIFFIES_SHIFT

#define NSEC_PER_JIFFY  ((NSEC_PER_SEC+HZ/2)/HZ)
#define NSEC_PER_SEC    1000000000L
```

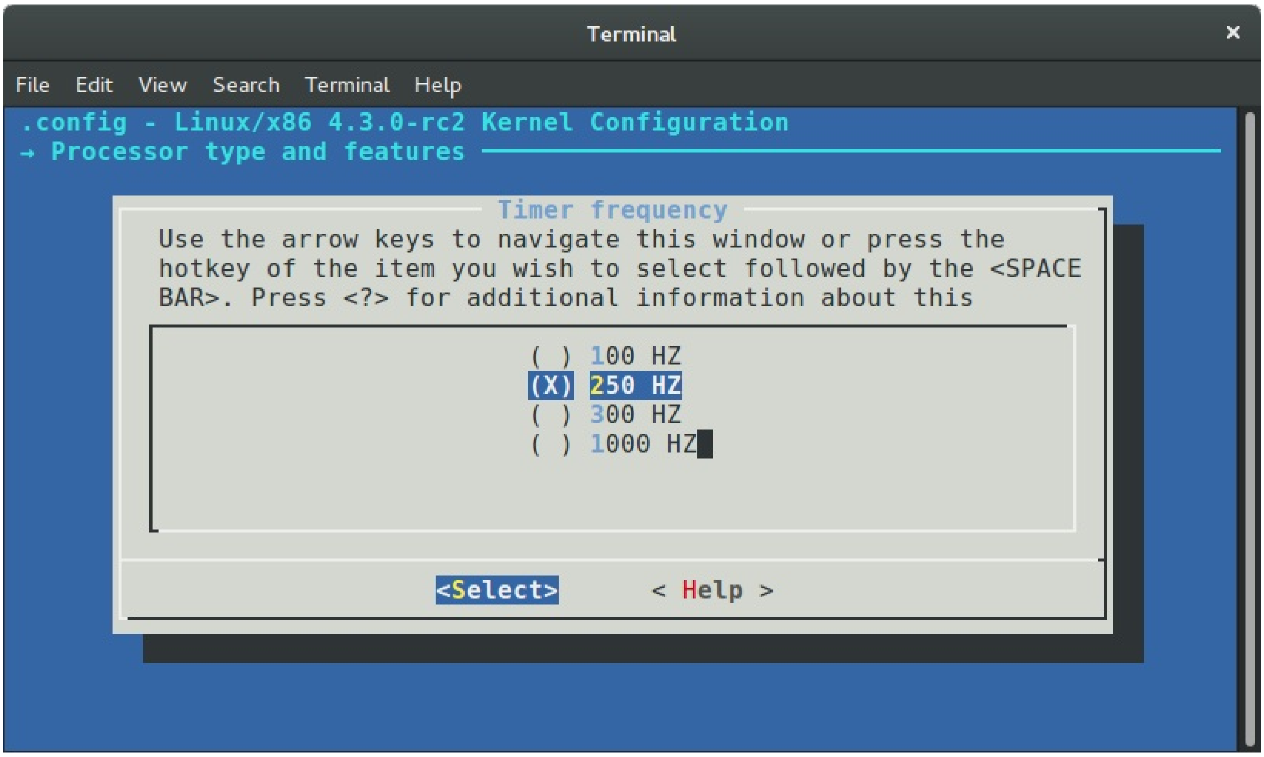
by default, and the `shift` is

```
#if HZ < 34
  #define JIFFIES_SHIFT   6
#elif HZ < 67
  #define JIFFIES_SHIFT   7
#else
  #define JIFFIES_SHIFT   8
#endif
```

The `jiffies` clock source uses the `NSEC_PER_JIFFY` multiplier conversion to specify the nanosecond over cycle ratio. Note that values of the `JIFFIES_SHIFT` and `NSEC_PER_JIFFY` depend on `HZ` value. The `HZ` represents the frequency of the system timer. This macro defined in the include/asm-generic/param.h and depends on the `CONFIG_HZ` kernel configuration option. The value of `HZ` differs for each supported architecture, but for `x86` it's defined like:

```
#define HZ        CONFIG_HZ
```

Where `CONFIG_HZ` can be one of the following values:



This means that in our case the timer interrupt frequency is `250 HZ` or occurs `250` times per second or one timer interrupt each `4ms` .

The last field that we can see in the definition of the `clocksource_jiffies` structure is the - `max_cycles` that holds the maximum cycle value that can safely be multiplied without potentially causing an overflow.

```
Ok, we just saw definition of the `clocksource_jiffies` structure, also we know a litt
le about `jiffies` and `clocksource`, now is time to get back to the implementation of
 the our function. In the beginning of this part we have stopped on the call of the:
```

```
register_refined_jiffies(CLOCK_TICK_RATE);
```

function from the arch/x86/kernel/setup.c source code file.

As I already wrote, the main purpose of the `register_refined_jiffies` function is to register `refined_jiffies` clocksource. We already saw the `clocksource_jiffies` structure represents standard `jiffies` clock source. Now, if you look in the kernel/time/jiffies.c source code file, you will find yet another clock source definition:

```
struct clocksource refined_jiffies;
```

There is one different between `refined_jiffies` and `clocksource_jiffies` : The standard `jiffies` based clock source is the lowest common denominator clock source which should function on all systems. As we already know, the `jiffies` global variable will be increased during each timer interrupt. This means that standard `jiffies` based clock source has the same resolution as the timer interrupt frequency. From this we can understand that standard `jiffies` based clock source may suffer from inaccuracies. The `refined_jiffies` uses `CLOCK_TICK_RATE` as the base of `jiffies` shift.

Let's look on the implementation of this function. First of all we can see that the `refined_jiffies` clock source based on the `clocksource_jiffies` structure:

```
int register_refined_jiffies(long cycles_per_second)
{
    u64 nsec_per_tick, shift_hz;
    long cycles_per_tick;

    refined_jiffies = clocksource_jiffies;
    refined_jiffies.name = "refined-jiffies";
    refined_jiffies.rating++;
    ...
    ...
    ...
```

Here we can see that we update the name of the `refined_jiffies` to `refined-jiffies` and increase the rating of this structure. As you remember, the `clocksource_jiffies` has rating - `1`, so our `refined_jiffies` clocksource will have rating - `2`. This means that the `refined_jiffies` will be best selection for clock source management code.

In the next step we need to calculate number of cycles per one tick:

```
cycles_per_tick = (cycles_per_second + HZ/2)/HZ;
```

Note that we have used `NSEC_PER_SEC` macro as the base of the standard `jiffies` multiplier. Here we are using the `cycles_per_second` which is the first parameter of the `register_refined_jiffies` function. We've passed the `CLOCK_TICK_RATE` macro to the `register_refined_jiffies` function. This macro definied in the arch/x86/include/asm/timex.h header file and expands to the:

```
#define CLOCK_TICK_RATE         PIT_TICK_RATE
```

where the `PIT_TICK_RATE` macro expands to the frequency of the Intel 8253:

```
#define PIT_TICK_RATE 1193182ul
```

After this we calculate `shift_hz` for the `register_refined_jiffies` that will store `hz << 8` or in other words frequency of the system timer. We shift left the `cycles_per_second` or frequency of the programmable interval timer on `8` in order to get extra accuracy:

```
shift_hz = (u64)cycles_per_second << 8;
shift_hz += cycles_per_tick/2;
do_div(shift_hz, cycles_per_tick);
```

In the next step we calculate the number of seconds per one tick by shifting left the `NSEC_PER_SEC` on `8` too as we did it with the `shift_hz` and do the same calculation as before:

```
nsec_per_tick = (u64)NSEC_PER_SEC << 8;
nsec_per_tick += (u32)shift_hz/2;
do_div(nsec_per_tick, (u32)shift_hz);
```

```
refined_jiffies.mult = ((u32)nsec_per_tick) << JIFFIES_SHIFT;
```

In the end of the `register_refined_jiffies` function we register new clock source with the `__clocksource_register` function that defined in the include/linux/clocksource.h header file and return:

```
__clocksource_register(&refined_jiffies);
return 0;
```

The clock source management code provides the API for clock source registration and selection. As we can see, clock sources are registered by calling the `__clocksource_register` function during kernel initialization or from a kernel module. During registration, the clock source management code will choose the best clock source available in the system using the `clocksource.rating` field which we already saw when we initialized `clocksource` structure for `jiffies`.

Using the jiffies

We just saw initialization of two `jiffies` based clock sources in the previous paragraph:

- standard `jiffies` based clock source;
- refined `jiffies` based clock source;

Don't worry if you don't understand the calculations here. They look frightening at first. Soon, step by step we will learn these things. So, we just saw initialization of `jffies` based clock sources and also we know that the Linux kernel has the global variable `jiffies` that holds the number of ticks that have occurred since the kernel started to work. Now, let's look how to use it. To use `jiffies` we just can use `jiffies` global variable by its name or with the call of the `get_jiffies_64` function. This function defined in the kernel/time/jiffies.c source code file and just returns full `64-bit` value of the `jiffies`:

```
u64 get_jiffies_64(void)
{
    unsigned long seq;
    u64 ret;

    do {
        seq = read_seqbegin(&jiffies_lock);
        ret = jiffies_64;
    } while (read_seqretry(&jiffies_lock, seq));
    return ret;
}
EXPORT_SYMBOL(get_jiffies_64);
```

Note that the `get_jiffies_64` function does not implemented as `jiffies_read` for example:

```
static cycle_t jiffies_read(struct clocksource *cs)
{
    return (cycle_t) jiffies;
}
```

We can see that implementation of the `get_jiffies_64` is more complex. The reading of the `jiffies_64` variable is implemented using seqlocks. Actually this is done for machines that cannot atomically read the full 64-bit values.

If we can access the `jiffies` or the `jiffies_64` variable we can convert it to `human` time units. To get one second we can use following expression:

```
jiffies / HZ
```

So, if we know this, we can get any time units. For example:

```
/* Thirty seconds from now */
jiffies + 30*HZ

/* Two minutes from now */
jiffies + 120*HZ

/* One millisecond from now */
jiffies + HZ / 1000
```

That's all.

Conclusion

This concludes the first part covering time and time management related concepts in the Linux kernel. We met first two concepts and its initialization in this part: `jiffies` and `clocksource`. In the next part we will continue to dive into this interesting theme and as I already wrote in this part we will acquainted and try to understand insides of these and other time management concepts in the Linux kernel.

If you have questions or suggestions, feel free to ping me in twitter 0xAX, drop me email or just create issue.

Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-insides.

Links

- system call
- TCP
- lock validator
- cgroups
- bss
- initrd
- Intel MID
- TSC
- void
- Simple Firmware Interface
- x86_64
- real time clock
- Jiffy
- high precision event timer
- nanoseconds
- Intel 8253
- seqlocks
- cloksource documentation
- Previous chapter

Timers and time management in the Linux kernel. Part 2.

Introduction to the `clocksource` framework

The previous part was the first part in the current chapter that describes timers and time management related stuff in the Linux kernel. We got acquainted with two concepts in the previous part:

* `jiffies`
* `clocksource`

The first is the global variable that is defined in the include/linux/jiffies.h header file and represents the counter that is increased during each timer interrupt. So if we can access this global variable and we know the timer interrupt rate we can convert `jiffies` to the human time units. As we already know the timer interrupt rate represented by the compile-time constant that is called `HZ` in the Linux kernel. The value of `HZ` is equal to the value of the `CONFIG_HZ` kernel configuration option and if we will look into the arch/x86/configs/x86_64_defconfig kernel configuration file, we will see that:

```
CONFIG_HZ_1000=y
```

kernel configuration option is set. This means that value of `CONFIG_HZ` will be `1000` by default for the x86_64 architecture. So, if we divide the value of `jiffies` by the value of `HZ` :

```
jiffies / HZ
```

we will get the amount of seconds that elapsed since the beginning of the moment the Linux kernel started to work or in other words we will get the system uptime. Since `HZ` represents the amount of timer interrupts in a second, we can set a value for some time in the future. For example:

```
/* one minute from now */
unsigned long later = jiffies + 60*HZ;

/* five minutes from now */
unsigned long later = jiffies + 5*60*HZ;
```

This is a very common practice in the Linux kernel. For example, if you will look into the arch/x86/kernel/smpboot.c source code file, you will find the `do_boot_cpu` function. This function boots all processors besides bootstrap processor. You can find a snippet that waits ten seconds for a response from the application processor:

```
if (!boot_error) {
    timeout = jiffies + 10*HZ;
    while (time_before(jiffies, timeout)) {
        ...
        ...
        ...
        udelay(100);
    }
    ...
    ...
    ...
}
```

We assign `jiffies + 10*HZ` value to the `timeout` variable here. As I think you already understood, this means a ten seconds timeout. After this we are entering a loop where we use the `time_before` macro to compare the current `jiffies` value and our timeout.

Or for example if we look into the sound/isa/sscape.c source code file which represents the driver for the Ensoniq Soundscape Elite sound card, we will see the `obp_startup_ack` function that waits upto a given timeout for the On-Board Processor to return its start-up acknowledgement sequence:

```
static int obp_startup_ack(struct soundscape *s, unsigned timeout)
{
    unsigned long end_time = jiffies + msecs_to_jiffies(timeout);

    do {
        ...
        ...
        ...
        x = host_read_unsafe(s->io_base);
        ...
        ...
        ...
        if (x == 0xfe || x == 0xff)
            return 1;
        msleep(10);
    } while (time_before(jiffies, end_time));

    return 0;
}
```

As you can see, the `jiffies` variable is very widely used in the Linux kernel code. As I already wrote, we met yet another new time management related concept in the previous part - `clocksource`. We have only seen a short description of this concept and the API for a clock source registration. Let's take a closer look in this part.

Introduction to `clocksource`

The `clocksource` concept represents the generic API for clock sources management in the Linux kernel. Why do we need a separate framework for this? Let's go back to the beginning. The `time` concept is the fundamental concept in the Linux kernel and other operating system kernels. And the timekeeping is one of the necessities to use this concept. For example Linux kernel must know and update the time elapsed since system startup, it must determine how long the current process has been running for every processor and many many more. Where the Linux kernel can get information about time? First of all it is Real Time Clock or RTC that represents by the a nonvolatile device. You can find a set of architecture-independent real time clock drivers in the Linux kernel in the drivers/rtc directory. Besides this, each architecture can provide a driver for the architecture-dependent real time clock, for example - `CMOS/RTC` - arch/x86/kernel/rtc.c for the x86 architecture. The second is system timer - timer that excites interrupts with a periodic rate. For example, for IBM PC compatibles it was - programmable interval timer.

We already know that for timekeeping purposes we can use `jiffies` in the Linux kernel. The `jiffies` can be considered as read only global variable which is updated with `HZ` frequency. We know that the `HZ` is a compile-time kernel parameter whose reasonable range is from `100` to `1000` Hz. So, it is guaranteed to have an interface for time measurement with `1` - `10` milliseconds resolution. Besides standard `jiffies`, we saw the `refined_jiffies` clock source in the previous part that is based on the `i8253/i8254` programmable interval timer tick rate which is almost `1193182` hertz. So we can get something about `1` microsecond resolution with the `refined_jiffies`. In this time, nanoseconds are the favorite choice for the time value units of the given clock source.

The availability of more precise techniques for time intervals measurement is hardware-dependent. We just knew a little about `x86` dependent timers hardware. But each architecture provides own timers hardware. Earlier each architecture had own implementation for this purpose. Solution of this problem is an abstraction layer and associated API in a common code framework for managing various clock sources and independent of the timer interrupt. This common code framework became - `clocksource` framework.

Generic timeofday and clock source management framework moved a lot of timekeeping code into the architecture independent portion of the code, with the architecture-dependent portion reduced to defining and managing low-level hardware pieces of clocksources. It takes a large amount of funds to measure the time interval on different architectures with different hardware, and it is very complex. Implementation of the each clock related service is strongly associated with an individual hardware device and as you can understand, it results in similar implementations for different architectures.

Within this framework, each clock source is required to maintain a representation of time as a monotonically increasing value. As we can see in the Linux kernel code, nanoseconds are the favorite choice for the time value units of a clock source in this time. One of the main point of the clock source framework is to allow an user to select clock source among a range of available hardware devices supporting clock functions when configuring the system and selecting, accessing and scaling different clock sources.

The clocksource structure

The fundamental of the `clocksource` framework is the `clocksource` structure that defined in the include/linux/clocksource.h header file. We already saw some fields that are provided by the `clocksource` structure in the previous part. Let's look on the full definition of this structure and try to describe all of its fields:

```
struct clocksource {
    cycle_t (*read)(struct clocksource *cs);
    cycle_t mask;
    u32 mult;
    u32 shift;
    u64 max_idle_ns;
    u32 maxadj;
#ifdef CONFIG_ARCH_CLOCKSOURCE_DATA
    struct arch_clocksource_data archdata;
#endif
    u64 max_cycles;
    const char *name;
    struct list_head list;
    int rating;
    int (*enable)(struct clocksource *cs);
    void (*disable)(struct clocksource *cs);
    unsigned long flags;
    void (*suspend)(struct clocksource *cs);
    void (*resume)(struct clocksource *cs);
#ifdef CONFIG_CLOCKSOURCE_WATCHDOG
    struct list_head wd_list;
    cycle_t cs_last;
    cycle_t wd_last;
#endif
    struct module *owner;
} ____cacheline_aligned;
```

We already saw the first field of the `clocksource` structure in the previous part - it is pointer to the `read` function that returns best counter selected by the clocksource framework. For example we use `jiffies_read` function to read `jiffies` value:

```
static struct clocksource clocksource_jiffies = {
    ...
    .read        = jiffies_read,
    ...
}
```

where `jiffies_read` just returns:

```
static cycle_t jiffies_read(struct clocksource *cs)
{
    return (cycle_t) jiffies;
}
```

Or the `read_tsc` function:

```
static struct clocksource clocksource_tsc = {
    ...
    .read                   = read_tsc,
    ...
};
```

for the time stamp counter reading.

The next field is `mask` that allows to ensure that subtraction between counters values from non `64 bit` counters do not need special overflow logic. After the `mask` field, we can see two fields: `mult` and `shift`. These are the fields that are base of mathematical functions that are provide ability to convert time values specific to each clock source. In other words these two fields help us to convert an abstract machine time units of a counter to nanoseconds.

After these two fields we can see the `64` bits `max_idle_ns` field represents max idle time permitted by the clocksource in nanoseconds. We need in this field for the Linux kernel with enabled `CONFIG_NO_HZ` kernel configuration option. This kernel configuration option enables the Linux kernel to run without a regular timer tick (we will see full explanation of this in other part). The problem that dynamic tick allows the kernel to sleep for periods longer than a single tick, moreover sleep time could be unlimited. The `max_idle_ns` field represents this sleeping limit.

The next field after the `max_idle_ns` is the `maxadj` field which is the maximum adjustment value to `mult`. The main formula by which we convert cycles to the nanoseconds:

```
((u64) cycles * mult) >> shift;
```

is not `100%` accurate. Instead the number is taken as close as possible to a nanosecond and `maxadj` helps to correct this and allows clocksource API to avoid `mult` values that might overflow when adjusted. The next four fields are pointers to the function:

- `enable` - optional function to enable clocksource;
- `disable` - optional function to disable clocksource;
- `suspend` - suspend function for the clocksource;
- `resume` - resume function for the clocksource;

The next field is the `max_cycles` and as we can understand from its name, this field represents maximum cycle value before potential overflow. And the last field is `owner` represents reference to a kernel module that is owner of a clocksource. This is all. We just went through all the standard fields of the `clocksource` structure. But you can noted that we missed some fields of the `clocksource` structure. We can divide all of missed field on two types: Fields of the first type are already known for us. For example, they are `name` field

that represents name of a `clocksource`, the `rating` field that helps to the Linux kernel to select the best clocksource and etc. The second type, fields which are dependent from the different Linux kernel configuration options. Let's look on these fields.

The first field is the `archdata`. This field has `arch_clocksource_data` type and depends on the `CONFIG_ARCH_CLOCKSOURCE_DATA` kernel configuration option. This field is actual only for the x86 and IA64 architectures for this moment. And again, as we can understand from the field's name, it represents architecture-specific data for a clock source. For example, it represents `vDSO` clock mode:

```
struct arch_clocksource_data {
    int vclock_mode;
};
```

for the `x86` architectures. Where the `vDSO` clock mode can be one of the:

```
#define VCLOCK_NONE 0
#define VCLOCK_TSC  1
#define VCLOCK_HPET 2
#define VCLOCK_PVCLOCK 3
```

The last three fields are `wd_list`, `cs_last` and the `wd_last` depends on the `CONFIG_CLOCKSOURCE_WATCHDOG` kernel configuration option. First of all let's try to understand what is it `watchdog`. In a simple words, watchdog is a timer that is used for detection of the computer malfunctions and recovering from it. All of these three fields contain watchdog related data that is used by the `clocksource` framework. If we will grep the Linux kernel source code, we will see that only arch/x86/KConfig kernel configuration file contains the `CONFIG_CLOCKSOURCE_WATCHDOG` kernel configuration option. So, why do `x86` and `x86_64` need in watchdog? You already may know that all `x86` processors has special 64-bit register - time stamp counter. This register contains number of cycles since the reset. Sometimes the time stamp counter needs to be verified against another clock source. We will not see initialization of the `watchdog` timer in this part, before this we must learn more about timers.

That's all. From this moment we know all fields of the `clocksource` structure. This knowledge will help us to learn insides of the `clocksource` framework.

New clock source registration

We saw only one function from the `clocksource` framework in the previous part. This function was - `__clocksource_register`. This function defined in the include/linux/clocksource.h header file and as we can understand from the function's name, main point of this function is to register new clocksource. If we will look on the implementation of the `__clocksource_register` function, we will see that it just makes call of the `__clocksource_register_scale` function and returns its result:

```
static inline int __clocksource_register(struct clocksource *cs)
{
    return __clocksource_register_scale(cs, 1, 0);
}
```

Before we will see implementation of the `__clocksource_register_scale` function, we can see that `clocksource` provides additional API for a new clock source registration:

```
static inline int clocksource_register_hz(struct clocksource *cs, u32 hz)
{
        return __clocksource_register_scale(cs, 1, hz);
}

static inline int clocksource_register_khz(struct clocksource *cs, u32 khz)
{
        return __clocksource_register_scale(cs, 1000, khz);
}
```

And all of these functions do the same. They return value of the `__clocksource_register_scale` function but with different set of parameters. The `__clocksource_register_scale` function defined in the kernel/time/clocksource.c source code file. To understand difference between these functions, let's look on the parameters of the `clocksource_register_khz` function. As we can see, this function takes three parameters:

- `cs` - clocksource to be installed;
- `scale` - scale factor of a clock source. In other words, if we will multiply value of this parameter on frequency, we will get `hz` of a clocksource;
- `freq` - clock source frequency divided by scale.

Now let's look on the implementation of the `__clocksource_register_scale` function:

```
int __clocksource_register_scale(struct clocksource *cs, u32 scale, u32 freq)
{
        __clocksource_update_freq_scale(cs, scale, freq);
        mutex_lock(&clocksource_mutex);
        clocksource_enqueue(cs);
        clocksource_enqueue_watchdog(cs);
        clocksource_select();
        mutex_unlock(&clocksource_mutex);
        return 0;
}
```

First of all we can see that the `__clocksource_register_scale` function starts from the call of the `__clocksource_update_freq_scale` function that defined in the same source code file and updates given clock source with the new frequency. Let's look on the implementation of this function. In the first step we need to check given frequency and if it was not passed as `zero`, we need to calculate `mult` and `shift` parameters for the given clock source. Why do we need to check value of the `frequency`? Actually it can be zero. if you attentively looked on the implementation of the `__clocksource_register` function, you may have noticed that we passed `frequency` as `0`. We will do it only for some clock sources that have self defined `mult` and `shift` parameters. Look in the previous part and you will see that we saw calculation of the `mult` and `shift` for `jiffies`. The `__clocksource_update_freq_scale` function will do it for us for other clock sources.

So in the start of the `__clocksource_update_freq_scale` function we check the value of the `frequency` parameter and if is not zero we need to calculate `mult` and `shift` for the given clock source. Let's look on the `mult` and `shift` calculation:

```
void __clocksource_update_freq_scale(struct clocksource *cs, u32 scale, u32 freq)
{
        u64 sec;

        if (freq) {
             sec = cs->mask;
             do_div(sec, freq);
             do_div(sec, scale);

             if (!sec)
                   sec = 1;
             else if (sec > 600 && cs->mask > UINT_MAX)
                   sec = 600;

             clocks_calc_mult_shift(&cs->mult, &cs->shift, freq,
                                   NSEC_PER_SEC / scale, sec * scale);
        }
        ...
        ...
        ...
}
```

Here we can see calculation of the maximum number of seconds which we can run before a clock source counter will overflow. First of all we fill the `sec` variable with the value of a clock source mask. Remember that a clock source's mask represents maximum amount of bits that are valid for the given clock source. After this, we can see two division operations. At first we divide our `sec` variable on a clock source frequency and then on scale factor. The `freq` parameter shows us how many timer interrupts will be occurred in one second. So, we divide `mask` value that represents maximum number of a counter (for example `jiffy`) on the frequency of a timer and will get the maximum number of seconds for the certain clock source. The second division operation will give us maximum number of seconds for the certain clock source depends on its scale factor which can be `1` hertz or `1` kilohertz (10^ Hz).

After we have got maximum number of seconds, we check this value and set it to `1` or `600` depends on the result at the next step. These values is maximum sleeping time for a clocksource in seconds. In the next step we can see call of the `clocks_calc_mult_shift`. Main point of this function is calculation of the `mult` and `shift` values for a given clock source. In the end of the `__clocksource_update_freq_scale` function we check that just calculated `mult` value of a given clock source will not cause overflow after adjustment, update the `max_idle_ns` and `max_cycles` values of a given clock source with the maximum nanoseconds that can be converted to a clock source counter and print result to the kernel buffer:

```
pr_info("%s: mask: 0x%llx max_cycles: 0x%llx, max_idle_ns: %lld ns\n",
    cs->name, cs->mask, cs->max_cycles, cs->max_idle_ns);
```

that we can see in the dmesg output:

```
$ dmesg | grep "clocksource:"
[    0.000000] clocksource: refined-jiffies: mask: 0xffffffff max_cycles: 0xffffffff,
max_idle_ns: 1910969940391419 ns
[    0.000000] clocksource: hpet: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns
: 133484882848 ns
[    0.094084] clocksource: jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max_idle
_ns: 1911260446275000 ns
[    0.205302] clocksource: acpi_pm: mask: 0xffffff max_cycles: 0xffffff, max_idle_ns:
 2085701024 ns
[    1.452979] clocksource: tsc: mask: 0xffffffffffffffff max_cycles: 0x7350b459580, m
ax_idle_ns: 881591204237 ns
```

After the `__clocksource_update_freq_scale` function will finish its work, we can return back to the `__clocksource_register_scale` function that will register new clock source. We can see the call of the following three functions:

```
mutex_lock(&clocksource_mutex);
clocksource_enqueue(cs);
clocksource_enqueue_watchdog(cs);
clocksource_select();
mutex_unlock(&clocksource_mutex);
```

Note that before the first will be called, we lock the `clocksource_mutex` mutex. The point of the `clocksource_mutex` mutex is to protect `curr_clocksource` variable which represents currently selected `clocksource` and `clocksource_list` variable which represents list that contains registered `clocksources`. Now, let's look on these three functions.

The first `clocksource_enqueue` function and other two defined in the same source code file. We go through all already registered `clocksources` or in other words we go through all elements of the `clocksource_list` and tries to find best place for a given `clocksource`:

```
static void clocksource_enqueue(struct clocksource *cs)
{
    struct list_head *entry = &clocksource_list;
    struct clocksource *tmp;

    list_for_each_entry(tmp, &clocksource_list, list)
        if (tmp->rating >= cs->rating)
            entry = &tmp->list;
    list_add(&cs->list, entry);
}
```

In the end we just insert new clocksource to the `clocksource_list`. The second function - `clocksource_enqueue_watchdog` does almost the same that previous function, but it inserts new clock source to the `wd_list` depends on flags of a clock source and starts new watchdog timer. As I already wrote, we will not consider `watchdog` related stuff in this part but will do it in next parts.

The last function is the `clocksource_select`. As we can understand from the function's name, main point of this function - select the best `clocksource` from registered clocksources. This function consists only from the call of the function helper:

```
static void clocksource_select(void)
{
    return __clocksource_select(false);
}
```

Note that the `__clocksource_select` function takes one parameter (`false` in our case). This bool parameter shows how to traverse the `clocksource_list`. In our case we pass `false` that is meant that we will go through all entries of the `clocksource_list`. We already know that `clocksource` with the best rating will the first in the `clocksource_list` after the call of the `clocksource_enqueue` function, so we can easily get it from this list. After we found a clock source with the best rating, we switch to it:

```
if (curr_clocksource != best && !timekeeping_notify(best)) {
    pr_info("Switched to clocksource %s\n", best->name);
    curr_clocksource = best;
}
```

The result of this operation we can see in the `dmesg` output:

```
$ dmesg | grep Switched
[    0.199688] clocksource: Switched to clocksource hpet
[    2.452966] clocksource: Switched to clocksource tsc
```

Note that we can see two clock sources in the `dmesg` output (`hpet` and `tsc` in our case). Yes, actually there can be many different clock sources on a particular hardware. So the Linux kernel knows about all registered clock sources and switches to a clock source with a better rating each time after registration of a new clock source.

If we will look on the bottom of the kernel/time/clocksource.c source code file, we will see that it has sysfs interface. Main initialization occurs in the `init_clocksource_sysfs` function which will be called during device `initcalls`. Let's look on the implementation of the `init_clocksource_sysfs` function:

```
static struct bus_type clocksource_subsys = {
    .name = "clocksource",
    .dev_name = "clocksource",
};

static int __init init_clocksource_sysfs(void)
{
    int error = subsys_system_register(&clocksource_subsys, NULL);

    if (!error)
        error = device_register(&device_clocksource);
    if (!error)
        error = device_create_file(
                &device_clocksource,
                &dev_attr_current_clocksource);
    if (!error)
        error = device_create_file(&device_clocksource,
                       &dev_attr_unbind_clocksource);
    if (!error)
        error = device_create_file(
                &device_clocksource,
                &dev_attr_available_clocksource);
    return error;
}
device_initcall(init_clocksource_sysfs);
```

First of all we can see that it registers a `clocksource` subsystem with the call of the `subsys_system_register` function. In other words, after the call of this function, we will have following directory:

```
$ pwd
/sys/devices/system/clocksource
```

After this step, we can see registration of the `device_clocksource` device which is represented by the following structure:

```
static struct device device_clocksource = {
    .id     = 0,
    .bus    = &clocksource_subsys,
};
```

and creation of three files:

- `dev_attr_current_clocksource` ;
- `dev_attr_unbind_clocksource` ;
- `dev_attr_available_clocksource` .

These files will provide information about current clock source in the system, available clock sources in the system and interface which allows to unbind the clock source.

After the `init_clocksource_sysfs` function will be executed, we will be able find some information about available clock sources in the:

```
$ cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
```

Or for example information about current clock source in the system:

```
$ cat /sys/devices/system/clocksource/clocksource0/current_clocksource
tsc
```

In the previous part, we saw API for the registration of the `jiffies` clock source, but didn't dive into details about the `clocksource` framework. In this part we did it and saw implementation of the new clock source registration and selection of a clock source with the best rating value in the system. Of course, this is not all API that `clocksource` framework provides. There a couple additional functions like `clocksource_unregister` for removing given clock source from the `clocksource_list` and etc. But I will not describe this functions in this part, because they are not important for us right now. Anyway if you are interesting in it, you can find it in the kernel/time/clocksource.c.

That's all.

Conclusion

This is the end of the second part of the chapter that describes timers and timer management related stuff in the Linux kernel. In the previous part got acquainted with the following two concepts: `jiffies` and `clocksource` . In this part we saw some examples of the `jiffies` usage and knew more details about the `clocksource` concept.

If you have questions or suggestions, feel free to ping me in twitter 0xAX, drop me email or just create issue.

Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-insides.

Links

- x86
- x86_64
- uptime
- Ensoniq Soundscape Elite
- RTC
- interrupts
- IBM PC
- programmable interval timer
- Hz
- nanoseconds
- dmesg
- time stamp counter
- loadable kernel module
- IA64
- watchdog
- clock rate
- mutex
- sysfs
- previous part

Timers and time management in the Linux kernel. Part 3.

The tick broadcast framework and dyntick

This is third part of the chapter which describes timers and time management related stuff in the Linux kernel and we stopped on the `clocksource` framework in the previous part. We have started to consider this framework because it is closely related to the special counters which are provided by the Linux kernel. One of these counters which we already saw in the first part of this chapter is - `jiffies`. As I already wrote in the first part of this chapter, we will consider time management related stuff step by step during the Linux kernel initialization. Previous step was call of the:

```
register_refined_jiffies(CLOCK_TICK_RATE);
```

function which defined in the kernel/time/jiffies.c source code file and executes initialization of the `refined_jiffies` clock source for us. Recall that this function is called from the `setup_arch` function that defined in the https://github.com/torvalds/linux/blob/master/arch/x86/kernel/setup.c source code and executes architecture-specific (x86_64 in our case) initialization. Look on the implementation of the `setup_arch` and you will note that the call of the `register_refined_jiffies` is the last step before the `setup_arch` function will finish its work.

There are many different `x86_64` specific things already configured after the end of the `setup_arch` execution. For example some early interrupt handlers already able to handle interrupts, memory space reserved for the initrd, DMI scanned, the Linux kernel log buffer is already set and this means that the printk function is able to work, e820 parsed and the Linux kernel already knows about available memory and and many many other architecture specific things (if you are interesting, you can read more about the `setup_arch` function and Linux kernel initialization process in the second chapter of this book).

Now, the `setup_arch` finished its work and we can back to the generic Linux kernel code. Recall that the `setup_arch` function was called from the `start_kernel` function which is defined in the init/main.c source code file. So, we shall return to this function. You can see that there are many different function are called right after `setup_arch` function inside of the `start_kernel` function, but since our chapter is devoted to timers and time management related stuff, we will skip all code which is not related to this topic. The first function which is related to the time management in the Linux kernel is:

```
tick_init();
```

in the `start_kernel`. The `tick_init` function defined in the kernel/time/tick-common.c source code file and does two things:

* Initialization of `tick broadcast` framework related data structures;
* Initialization of `full` tickless mode related data structures.

We didn't see anything related to the `tick broadcast` framework in this book and didn't know anything about tickless mode in the Linux kernel. So, the main point of this part is to look on these concepts and to know what are they.

The idle process

First of all, let's look on the implementation of the `tick_init` function. As I already wrote, this function defined in the kernel/time/tick-common.c source code file and consists from the two calls of following functions:

```
void __init tick_init(void)
{
	tick_broadcast_init();
	tick_nohz_init();
}
```

As you can understand from the paragraph's title, we are interesting only in the `tick_broadcast_init` function for now. This function defined in the kernel/time/tick-broadcast.c source code file and executes initialization of the `tick broadcast` framework related data structures. Before we will look on the implementation of the `tick_broadcast_init` function and will try to understand what does this function do, we need to know about `tick broadcast` framework.

Main point of a central processor is to execute programs. But sometimes a processor may be in a special state when it is not being used by any program. This special state is called - idle. When the processor has no anything to execute, the Linux kernel launches `idle` task. We already saw a little about this in the last part of the Linux kernel initialization process. When the Linux kernel will finish all initialization processes in the `start_kernel` function from the init/main.c source code file, it will call the `rest_init` function from the same source code file. Main point of this function is to launch kernel `init` thread and the `kthreadd` thread, to call the `schedule` function to start task scheduling and to go to sleep by calling the `cpu_idle_loop` function that defined in the kernel/sched/idle.c source code file.

The `cpu_idle_loop` function represents infinite loop which checks the need for rescheduling on each iteration. After the scheduler finds something to execute, the `idle` process will finish its work and the control will be moved to a new runnable task with the call of the `schedule_preempt_disabled` function:

```
static void cpu_idle_loop(void)
{
    while (1) {
        while (!need_resched()) {
        ...
        ...
        ...
        /* the main idle function */
        cpuidle_idle_call();
    }
    ...
    ...
    ...
    schedule_preempt_disabled();
}
```

Of course, we will not consider full implementation of the `cpu_idle_loop` function and details of the `idle` state in this part, because it is not related to our topic. But there is one interesting moment for us. We know that the processor can execute only one task in one time. How does the Linux kernel decide to reschedule and stop `idle` process if the processor executes infinite loop in the `cpu_idle_loop`? The answer is system timer interrupts. When an interrupt occurs, the processor stops the `idle` thread and transfers control to an interrupt handler. After the system timer interrupt handler will be handled, the `need_resched` will return true and the Linux kernel will stop `idle` process and will transfer control to the current runnable task. But handling of the system timer interrupts is not effective for power management, because if a processor is in `idle` state, there is little point in sending it a system timer interrupt.

By default, there is the `CONFIG_HZ_PERIODIC` kernel configuration option which is enabled in the Linux kernel and tells to handle each interrupt of the system timer. To solve this problem, the Linux kernel provides two additional ways of managing scheduling-clock interrupts:

The first is to omit scheduling-clock ticks on idle processors. To enable this behaviour in the Linux kernel, we need to enable the `CONFIG_NO_HZ_IDLE` kernel configuration option. This option allows Linux kernel to avoid sending timer interrupts to idle processors. In this case periodic timer interrupts will be replaced with on-demand interrupts. This mode is called - `dyntick-idle` mode. But if the kernel does not handle interrupts of a system timer, how can the kernel decide if the system has nothing to do?

Whenever the idle task is selected to run, the periodic tick is disabled with the call of the `tick_nohz_idle_enter` function that defined in the kernel/time/tick-sched.c source code file and enabled with the call of the `tick_nohz_idle_exit` function. There is special concept in the Linux kernel which is called - `clock event devices` that are used to schedule the next interrupt. This concept provides API for devices which can deliver interrupts at a specific time in the future and represented by the `clock_event_device` structure in the Linux kernel. We will not dive into implementation of the `clock_event_device` structure now. We will see it in the next prat of this chapter. But there is one interesting moment for us right now.

The second way is to omit scheduling-clock ticks on processors that are either in `idle` state or that have only one runnable task or in other words busy processor. We can enable this feature with the `CONFIG_NO_HZ_FULL` kernel configuration option and it allows to reduce the number of timer interrupts significantly.

Besides the `cpu_idle_loop`, idle processor can be in a sleeping state. The Linux kernel provides special `cpuidle` framework. Main point of this framework is to put an idle processor to sleeping states. The name of the set of these states is - `C-states`. But how does a processor will be woken if local timer is disabled? The linux kernel provides `tick broadcast` framework for this. The main point of this framework is assign a timer which is not affected by the `C-states`. This timer will wake a sleeping processor.

Now, after some theory we can return to the implementation of our function. Let's recall that the `tick_init` function just calls two following functions:

```
void __init tick_init(void)
{
    tick_broadcast_init();
    tick_nohz_init();
}
```

Let's consider the first function. The first `tick_broadcast_init` function defined in the kernel/time/tick-broadcast.c source code file and executes initialization of the `tick broadcast` framework related data structures. Let's look on the implementation of the `tick_broadcast_init` function:

```
void __init tick_broadcast_init(void)
{
        zalloc_cpumask_var(&tick_broadcast_mask, GFP_NOWAIT);
        zalloc_cpumask_var(&tick_broadcast_on, GFP_NOWAIT);
        zalloc_cpumask_var(&tmpmask, GFP_NOWAIT);
#ifdef CONFIG_TICK_ONESHOT
         zalloc_cpumask_var(&tick_broadcast_oneshot_mask, GFP_NOWAIT);
         zalloc_cpumask_var(&tick_broadcast_pending_mask, GFP_NOWAIT);
         zalloc_cpumask_var(&tick_broadcast_force_mask, GFP_NOWAIT);
#endif
}
```

As we can see, the `tick_broadcast_init` function allocates different cpumasks with the help of the `zalloc_cpumask_var` function. The `zalloc_cpumask_var` function defined in the lib/cpumask.c source code file and expands to the call of the following function:

```
bool zalloc_cpumask_var(cpumask_var_t *mask, gfp_t flags)
{
        return alloc_cpumask_var(mask, flags | __GFP_ZERO);
}
```

Ultimately, the memory space will be allocated for the given `cpumask` with the certain flags with the help of the `kmalloc_node` function:

```
*mask = kmalloc_node(cpumask_size(), flags, node);
```

Now let's look on the `cpumasks` that will be initialized in the `tick_broadcast_init` function. As we can see, the `tick_broadcast_init` function will initialize six `cpumasks`, and moreover, initialization of the last three `cpumasks` will be depended on the `CONFIG_TICK_ONESHOT` kernel configuration option.

The first three `cpumasks` are:

- `tick_broadcast_mask` - the bitmap which represents list of processors that are in a sleeping mode;
- `tick_broadcast_on` - the bitmap that stores numbers of processors which are in a periodic broadcast state;
- `tmpmask` - this bitmap for temporary usage.

As we already know, the next three `cpumasks` depends on the `CONFIG_TICK_ONESHOT` kernel configuration option. Actually each clock event devices can be in one of two modes:

- `periodic` - clock events devices that support periodic events;
- `oneshot` - clock events devices that capable of issuing events that happen only once.

The linux kernel defines two mask for such clock events devices in the include/linux/clockchips.h header file:

```
#define CLOCK_EVT_FEAT_PERIODIC        0x000001
#define CLOCK_EVT_FEAT_ONESHOT         0x000002
```

So, the last three `cpumasks` are:

- `tick_broadcast_oneshot_mask` - stores numbers of processors that must be notified;
- `tick_broadcast_pending_mask` - stores numbers of processors that pending broadcast;
- `tick_broadcast_force_mask` - stores numbers of processors with enforced broadcast.

We have initialized six `cpumasks` in the `tick broadcast` framework, and now we can proceed to implementation of this framework.

The `tick broadcast` framework

Hardware may provide some clock source devices. When a processor sleeps and its local timer stopped, there must be additional clock source device that will handle awakening of a processor. The Linux kernel uses these `special` clock source devices which can raise an interrupt at a specified time. We already know that such timers called `clock events` devices in the Linux kernel. Besides `clock events` devices. Actually, each processor in the system has its own local timer which is programmed to issue interrupt at the time of the next deferred task. Also these timers can be programmed to do a periodical job, like updating `jiffies` and etc. These timers represented by the `tick_device` structure in the Linux kernel. This structure defined in the kernel/time/tick-sched.h header file and looks:

```
struct tick_device {
        struct clock_event_device *evtdev;
        enum tick_device_mode mode;
};
```

Note, that the `tick_device` structure contains two fields. The first field - `evtdev` represents pointer to the `clock_event_device` structure that defined in the include/linux/clockchips.h header file and represents descriptor of a clock event device. A `clock event` device allows to register an event that will happen in the future. As I already wrote, we will not consider `clock_event_device` structure and related API in this part, but will see it in the next part.

The second field of the `tick_device` structure represents mode of the `tick_device`. As we already know, the mode can be one of the:

```
num tick_device_mode {
        TICKDEV_MODE_PERIODIC,
        TICKDEV_MODE_ONESHOT,
};
```

Each `clock events` device in the system registers itself by the call of the `clockevents_register_device` function or `clockevents_config_and_register` function during initialization process of the Linux kernel. During the registration of a new `clock events` device, the Linux kernel calls the `tick_check_new_device` function that defined in the kernel/time/tick-common.c source code file and checks the given `clock events` device should be used by the Linux kernel. After all checks, the `tick_check_new_device` function executes a call of the:

```
tick_install_broadcast_device(newdev);
```

function that checks that the given `clock event` device can be broadcast device and install it, if the given device can be broadcast device. Let's look on the implementation of the `tick_install_broadcast_device` function:

```
void tick_install_broadcast_device(struct clock_event_device *dev)
{
    struct clock_event_device *cur = tick_broadcast_device.evtdev;

    if (!tick_check_broadcast_device(cur, dev))
        return;

    if (!try_module_get(dev->owner))
        return;

    clockevents_exchange_device(cur, dev);

    if (cur)
        cur->event_handler = clockevents_handle_noop;

    tick_broadcast_device.evtdev = dev;

    if (!cpumask_empty(tick_broadcast_mask))
        tick_broadcast_start_periodic(dev);

    if (dev->features & CLOCK_EVT_FEAT_ONESHOT)
        tick_clock_notify();
}
```

First of all we get the current `clock event` device from the `tick_broadcast_device`. The `tick_broadcast_device` defined in the kernel/time/tick-common.c source code file:

```
static struct tick_device tick_broadcast_device;
```

and represents external clock device that keeps track of events for a processor. The first step after we got the current clock device is the call of the `tick_check_broadcast_device` function which checks that a given clock events device can be utilized as broadcast device. The main point of the `tick_check_broadcast_device` function is to check value of the `features` field of the given `clock events` device. As we can understand from the name of this field, the `features` field contains a clock event device features. Available values defined in the include/linux/clockchips.h header file and can be one of the `CLOCK_EVT_FEAT_PERIODIC` - which represents a clock events device which supports periodic events and etc. So, the `tick_check_broadcast_device` function check `features` flags for `CLOCK_EVT_FEAT_ONESHOT` , `CLOCK_EVT_FEAT_DUMMY` and other flags and returns `false` if the given clock events device has one of these features. In other way the `tick_check_broadcast_device` function compares `ratings` of the given clock event device and current clock event device and returns the best.

After the `tick_check_broadcast_device` function, we can see the call of the `try_module_get` function that checks module owner of the clock events. We need to do it to be sure that the given `clock events` device was correctly initialized. The next step is the call of the `clockevents_exchange_device` function that defined in the kernel/time/clockevents.c source code file and will release old clock events device and replace the previous functional handler with a dummy handler.

In the last step of the `tick_install_broadcast_device` function we check that the `tick_broadcast_mask` is not empty and start the given `clock events` device in periodic mode with the call of the `tick_broadcast_start_periodic` function:

```
if (!cpumask_empty(tick_broadcast_mask))
    tick_broadcast_start_periodic(dev);

if (dev->features & CLOCK_EVT_FEAT_ONESHOT)
    tick_clock_notify();
```

The `tick_broadcast_mask` filled in the `tick_device_uses_broadcast` function that checks a `clock events` device during registration of this `clock events` device:

```
int cpu = smp_processor_id();

int tick_device_uses_broadcast(struct clock_event_device *dev, int cpu)
{
    ...
    ...
    ...
    if (!tick_device_is_functional(dev)) {
        ...
        cpumask_set_cpu(cpu, tick_broadcast_mask);
        ...
    }
    ...
    ...
    ...
}
```

More about the `smp_processor_id` macro you can read in the fourth part of the Linux kernel initialization process chapter.

The `tick_broadcast_start_periodic` function check the given `clock event` device and call the `tick_setup_periodic` function:

```
static void tick_broadcast_start_periodic(struct clock_event_device *bc)
{
    if (bc)
        tick_setup_periodic(bc, 1);
}
```

that defined in the kernel/time/tick-common.c source code file and sets broadcast handler for the given `clock event` device by the call of the following function:

```
tick_set_periodic_handler(dev, broadcast);
```

This function checks the second parameter which represents broadcast state (`on` or `off`) and sets the broadcast handler depends on its value:

```
void tick_set_periodic_handler(struct clock_event_device *dev, int broadcast)
{
    if (!broadcast)
        dev->event_handler = tick_handle_periodic;
    else
        dev->event_handler = tick_handle_periodic_broadcast;
}
```

When an `clock event` device will issue an interrupt, the `dev->event_handler` will be called. For example, let's look on the interrupt handler of the high precision event timer which is located in the arch/x86/kernel/hpet.c source code file:

```
static irqreturn_t hpet_interrupt_handler(int irq, void *data)
{
    struct hpet_dev *dev = (struct hpet_dev *)data;
    struct clock_event_device *hevt = &dev->evt;

    if (!hevt->event_handler) {
        printk(KERN_INFO "Spurious HPET timer interrupt on HPET timer %d\n",
                dev->num);
        return IRQ_HANDLED;
    }

    hevt->event_handler(hevt);
    return IRQ_HANDLED;
}
```

The `hpet_interrupt_handler` gets the irq specific data and check the event handler of the `clock event` device. Recall that we just set in the `tick_set_periodic_handler` function. So the `tick_handler_periodic_broadcast` function will be called in the end of the high precision event timer interrupt handler.

The `tick_handler_periodic_broadcast` function calls the

```
bc_local = tick_do_periodic_broadcast();
```

function which stores numbers of processors which have asked to be woken up in the temporary `cpumask` and call the `tick_do_broadcast` function:

```
cpumask_and(tmpmask, cpu_online_mask, tick_broadcast_mask);
return tick_do_broadcast(tmpmask);
```

The `tick_do_broadcast` calls the `broadcast` function of the given clock events which sends IPI interrupt to the set of the processors. In the end we can call the event handler of the given `tick_device` :

```
if (bc_local)
    td->evtdev->event_handler(td->evtdev);
```

which actually represents interrupt handler of the local timer of a processor. After this a processor will wake up. That is all about `tick broadcast` framework in the Linux kernel. We have missed some aspects of this framework, for example reprogramming of a `clock event`

device and broadcast with the oneshot timer and etc. But the Linux kernel is very big, it is not real to cover all aspects of it. I think it will be interesting to dive into with yourself.

If you remember, we have started this part with the call of the `tick_init` function. We just consider the `tick_broadcast_init` function and releated theory, but the `tick_init` function contains another call of a function and this function is - `tick_nohz_init`. Let's look on the implementation of this function.

Initialization of dyntick related data structures

We already saw some information about `dyntick` concept in this part and we know that this concept allows kernel to disable system timer interrupts in the `idle` state. The `tick_nohz_init` function makes initialization of the different data structures which are related to this concept. This function defined in the kernel/time/tick-sched.c source code file and starts from the check of the value of the `tick_nohz_full_running` variable which represents state of the tick-less mode for the `idle` state and the state when system timer interrups are disabled during a processor has only one runnable task:

```
if (!tick_nohz_full_running) {
    if (tick_nohz_init_all() < 0)
    return;
}
```

If this mode is not running we call the `tick_nohz_init_all` function that defined in the same source code file and check its result. The `tick_nohz_init_all` function tries to allocate the `tick_nohz_full_mask` with the call of the `alloc_cpumask_var` that will allocate space for a `tick_nohz_full_mask`. The `tck_nohz_full_mask` will store numbers of processors that have enabled full `NO_HZ`. After successful allocation of the `tick_nohz_full_mask` we set all bits in the `tick_nogz_full_mask`, set the `tick_nohz_full_running` and return result to the `tick_nohz_init` function:

```
static int tick_nohz_init_all(void)
{
        int err = -1;
#ifdef CONFIG_NO_HZ_FULL_ALL
        if (!alloc_cpumask_var(&tick_nohz_full_mask, GFP_KERNEL)) {
                WARN(1, "NO_HZ: Can't allocate full dynticks cpumask\n");
                return err;
        }
        err = 0;
        cpumask_setall(tick_nohz_full_mask);
        tick_nohz_full_running = true;
#endif
        return err;
}
```

In the next step we try to allocate a memory space for the `housekeeping_mask` :

```
if (!alloc_cpumask_var(&housekeeping_mask, GFP_KERNEL)) {
    WARN(1, "NO_HZ: Can't allocate not-full dynticks cpumask\n");
    cpumask_clear(tick_nohz_full_mask);
    tick_nohz_full_running = false;
    return;
}
```

This `cpumask` will store number of processor for `housekeeping` or in other words we need at least in one processor that will not be in `NO_HZ` mode, because it will do timekeeping and etc. After this we check the result of the architecture-specific `arch_irq_work_has_interrupt` function. This function checks ability to send inter-processor interrupt for the certain architecture. We need to check this, because system timer of a processor will be disabled during `NO_HZ` mode, so there must be at least one online processor which can send inter-processor interrupt to awake offline processor. This function defined in the arch/x86/include/asm/irq_work.h header file for the x86_64 and just checks that a processor has APIC from the CPUID:

```
static inline bool arch_irq_work_has_interrupt(void)
{
    return cpu_has_apic;
}
```

If a processor has not `APIC` , the Linux kernel prints warning message, clears the `tick_nohz_full_mask` cpumask, copies numbers of all possible processors in the system to the `housekeeping_mask` and resets the value of the `tick_nohz_full_running` variable:

```
if (!arch_irq_work_has_interrupt()) {
    pr_warning("NO_HZ: Can't run full dynticks because arch doesn't "
            "support irq work self-IPIs\n");
    cpumask_clear(tick_nohz_full_mask);
    cpumask_copy(housekeeping_mask, cpu_possible_mask);
    tick_nohz_full_running = false;
    return;
}
```

After this step, we get the number of the current processor by the call of the `smp_processor_id` and check this processor in the `tick_nohz_full_mask`. If the `tick_nohz_full_mask` contains a given processor we clear appropriate bit in the `tick_nohz_full_mask`:

```
cpu = smp_processor_id();

if (cpumask_test_cpu(cpu, tick_nohz_full_mask)) {
    pr_warning("NO_HZ: Clearing %d from nohz_full range for timekeeping\n", cpu);
    cpumask_clear_cpu(cpu, tick_nohz_full_mask);
}
```

Because this processor will be used for timekeeping. After this step we put all numbers of processors that are in the `cpu_possible_mask` and not in the `tick_nohz_full_mask`:

```
cpumask_andnot(housekeeping_mask,
           cpu_possible_mask, tick_nohz_full_mask);
```

After this operation, the `housekeeping_mask` will contain all processors of the system except a processor for timekeeping. In the last step of the `tick_nohz_init_all` function, we are going through all processors that are defined in the `tick_nohz_full_mask` and call the following function for an each processor:

```
for_each_cpu(cpu, tick_nohz_full_mask)
    context_tracking_cpu_set(cpu);
```

The `context_tracking_cpu_set` function defined in the kernel/context_tracking.c source code file and main point of this function is to set the `context_tracking.active` percpu variable to `true`. When the `active` field will be set to `true` for the certain processor, all context switches will be ignored by the Linux kernel context tracking subsystem for this processor.

That's all. This is the end of the `tick_nohz_init` function. After this `NO_HZ` related data structures will be initialzed. We didn't see API of the `NO_HZ` mode, but will see it soon.

Conclusion

This is the end of the third part of the chapter that describes timers and timer management related stuff in the Linux kernel. In the previous part got acquainted with the `clocksource` concept in the Linux kernel which represents framework for managing different clock source in a interrupt and hardware characteristics independent way. We continued to look on the Linux kernel initialization process in a time management context in this part and got acquainted with two new concepts for us: the `tick broadcast` framework and `tick-less` mode. The first concept helps the Linux kernel to deal with processors which are in deep sleep and the second concept represents the mode in which kernel may work to improve power management of `idle` processors.

In the next part we will continue to dive into timer management related things in the Linux kernel and will see new concept for us - `timers`.

If you have questions or suggestions, feel free to ping me in twitter 0xAX, drop me email or just create issue.

Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-insides.

Links

- x86_64
- initrd
- interrupt
- DMI
- printk
- CPU idle
- power management
- NO_HZ documentation
- cpumasks
- high precision event timer
- irq
- IPI
- CPUID
- APIC
- percpu
- context switches
- Previous part

Timers and time management in the Linux kernel. Part 4.

Timers

This is fourth part of the chapter which describes timers and time management related stuff in the Linux kernel and in the previous part we knew about the `tick broadcast` framework and `NO_HZ` mode in the Linux kernel. We will continue to dive into the time management related stuff in the Linux kernel in this part and will be acquainted with yet another concept in the Linux kernel - `timers`. Before we will look at timers in the Linux kernel, we have to learn some theory about this concept. Note that we will consider software timers in this part.

The Linux kernel provides a `software timer` concept to allow to kernel functions could be invoked at future moment. Timers are widely used in the Linux kernel. For example, look in the net/netfilter/ipset/ip_set_list_set.c source code file. This source code file provides implementation of the framework for the managing of groups of IP addresses.

We can find the `list_set` structure that contains `gc` filed in this source code file:

```
struct list_set {
    ...
    struct timer_list gc;
    ...
};
```

Not that the `gc` filed has `timer_list` type. This structure defined in the include/linux/timer.h header file and main point of this structure is to store `dynamic` timers in the Linux kernel. Actually, the Linux kernel provides two types of timers called dynamic timers and interval timers. First type of timers is used by the kernel, and the second can be used by user mode. The `timer_list` structure contains actual `dynamic` timers. The `list_set` contains `gc` timer in our example represents timer for garbage collection. This timer will be initialized in the `list_set_gc_init` function:

```
static void
list_set_gc_init(struct ip_set *set, void (*gc)(unsigned long ul_set))
{
    struct list_set *map = set->data;
    ...
    ...
    ...
    map->gc.function = gc;
    map->gc.expires = jiffies + IPSET_GC_PERIOD(set->timeout) * HZ;
    ...
    ...
    ...
}
```

A function that is pointed by the `gc` pointer, will be called after timeout which is equal to the `map->gc.expires`.

Ok, we will not dive into this example with the netfilter, because this chapter is not about network related stuff. But we saw that timers are widely used in the Linux kernel and learned that they represent concept which allows to functions to be called in future.

Now let's continue to research source code of Linux kernel which is related to the timers and time management stuff as we did it in all previous chapters.

Introduction to dynamic timers in the Linux kernel

As I already wrote, we knew about the `tick broadcast` framework and `NO_HZ` mode in the previous part. They will be initialized in the init/main.c source code file by the call of the `tick_init` function. If we will look at this source code file, we will see that the next time management related function is:

```
init_timers();
```

This function defined in the kernel/time/timer.c source code file and contains calls of four functions:

```
void __init init_timers(void)
{
    init_timer_cpus();
    init_timer_stats();
    timer_register_cpu_notifier();
    open_softirq(TIMER_SOFTIRQ, run_timer_softirq);
}
```

Let's look on implementation of each function. The first function is `init_timer_cpus` defined in the same source code file and just calls the `init_timer_cpu` function for each possible processor in the system:

```
static void __init init_timer_cpus(void)
{
    int cpu;

    for_each_possible_cpu(cpu)
        init_timer_cpu(cpu);
}
```

If you do not know or do not remember what is it a `possible` cpu, you can read the special part of this book which describes `cpumask` concept in the Linux kernel. In short words, a `possible` processor is a processor which can be plugged in anytime during the life of the system.

The `init_timer_cpu` function does main work for us, namely it executes initialization of the `tvec_base` structure for each processor. This structure defined in the kernel/time/timer.c source code file and stores data related to a `dynamic` timer for a certain processor. Let's look on the definition of this structure:

```
struct tvec_base {
    spinlock_t lock;
    struct timer_list *running_timer;
    unsigned long timer_jiffies;
    unsigned long next_timer;
    unsigned long active_timers;
    unsigned long all_timers;
    int cpu;
    bool migration_enabled;
    bool nohz_active;
    struct tvec_root tv1;
    struct tvec tv2;
    struct tvec tv3;
    struct tvec tv4;
    struct tvec tv5;
} ____cacheline_aligned;
```

The `thec_base` structure contains following fields: The `lock` for `tvec_base` protection, the next `running_timer` field points to the currently running timer for the certain processor, the `timer_jiffies` fields represents the earliest expiration time (it will be used by the Linux kernel to find already expired timers). The next field - `next_timer` contains the next pending timer for a next timer interrupt in a case when a processor goes to sleep and the `NO_HZ` mode is enabled in the Linux kernel. The `active_timers` field provides accounting of non-deferrable timers or in other words all timers that will not be stopped during a processor will go to sleep. The `all_timers` field tracks total number of timers or `active_timers` + deferrable timers. The `cpu` field represents number of a processor which owns timers. The `migration_enabled` and `nohz_active` fields are represent opportunity of timers migration to another processor and status of the `NO_HZ` mode respectively.

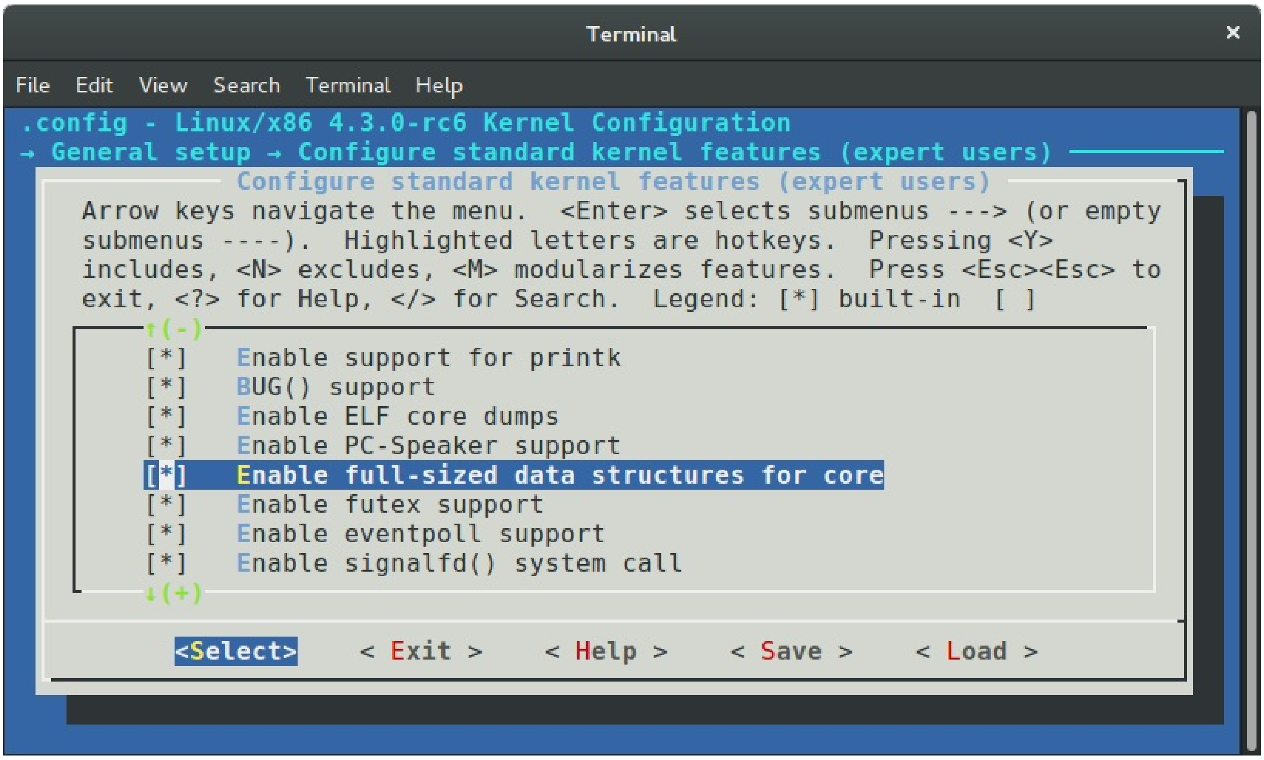
The last five fields of the `tvec_base` structure represent lists of dynamic timers. The first `tv1` field has:

```
#define TVR_SIZE (1 << TVR_BITS)
#define TVR_BITS (CONFIG_BASE_SMALL ? 6 : 8)

...
...
...

struct tvec_root {
    struct hlist_head vec[TVR_SIZE];
};
```

type. Note that the value of the `TVR_SIZE` depends on the `CONFIG_BASE_SMALL` kernel configuration option:



that reduces size of the kernel data structures if disabled. The `v1` is array that may contain `64` or `256` elements where an each element represents a dynamic timer that will decay within the next `255` system timer interrupts. Next three fields: `tv2` , `tv3` and `tv4` are lists with dynamic timers too, but they store dynamic timers which will decay the next `2^14 - 1` , `2^20 - 1` and `2^26` respectively. The last `tv5` field represents list which stores dynamic timers with a large expiring period.

So, now we saw the `tvec_base` structure and description of its fields and we can look on the implementation of the `init_timer_cpu` function. As I already wrote, this function defined in the kernel/time/timer.c source code file and executes initialization of the `tvec_bases` :

```
static void __init init_timer_cpu(int cpu)
{
    struct tvec_base *base = per_cpu_ptr(&tvec_bases, cpu);

    base->cpu = cpu;
    spin_lock_init(&base->lock);

    base->timer_jiffies = jiffies;
    base->next_timer = base->timer_jiffies;
}
```

The `tvec_bases` represents per-cpu variable which represents main data structure for a dynamic timer for a given processor. This `per-cpu` variable defined in the same source code file:

```
static DEFINE_PER_CPU(struct tvec_base, tvec_bases);
```

First of all we're getting the address of the `tvec_bases` for the given processor to `base` variable and as we got it, we are starting to initialize some of the `tvec_base` fields in the `init_timer_cpu` function. After initialization of the `per-cpu` dynamic timers with the jiffies and the number of a possible processor, we need to initialize a `tstats_lookup_lock` spinlock in the `init_timer_stats` function:

```
void __init init_timer_stats(void)
{
    int cpu;

    for_each_possible_cpu(cpu)
        raw_spin_lock_init(&per_cpu(tstats_lookup_lock, cpu));
}
```

The `tstats_lookcup_lock` variable represents `per-cpu` raw spinlock:

```
static DEFINE_PER_CPU(raw_spinlock_t, tstats_lookup_lock);
```

which will be used for protection of operation with statistics of timers that can be accessed through the procfs:

```
static int __init init_tstats_procfs(void)
{
    struct proc_dir_entry *pe;

    pe = proc_create("timer_stats", 0644, NULL, &tstats_fops);
    if (!pe)
        return -ENOMEM;
    return 0;
}
```

For example:

```
$ cat /proc/timer_stats
Timerstats sample period: 3.888770 s
  12,     0 swapper          hrtimer_stop_sched_tick (hrtimer_sched_tick)
  15,     1 swapper          hcd_submit_urb (rh_timer_func)
   4,   959 kedac            schedule_timeout (process_timeout)
   1,     0 swapper          page_writeback_init (wb_timer_fn)
  28,     0 swapper          hrtimer_stop_sched_tick (hrtimer_sched_tick)
  22,  2948 IRQ 4            tty_flip_buffer_push (delayed_work_timer_fn)
  ...
  ...
  ...
```

The next step after initialization of the `tstats_lookup_lock` spinlock is the call of the `timer_register_cpu_notifier` function. This function depends on the `CONFIG_HOTPLUG_CPU` kernel configuration option which enables support for hotplug processors in the Linux kernel.

When a processor will be logically offlined, a notification will be sent to the Linux kernel with the `CPU_DEAD` or the `CPU_DEAD_FROZEN` event by the call of the `cpu_notifier` macro:

```
#ifdef CONFIG_HOTPLUG_CPU
...
...
static inline void timer_register_cpu_notifier(void)
{
    cpu_notifier(timer_cpu_notify, 0);
}
...
...
#else
...
...
static inline void timer_register_cpu_notifier(void) { }
...
...
#endif /* CONFIG_HOTPLUG_CPU */
```

In this case the `timer_cpu_notify` will be called which checks an event type and will call the `migrate_timers` function:

```
static int timer_cpu_notify(struct notifier_block *self,
                            unsigned long action, void *hcpu)
{
    switch (action) {
    case CPU_DEAD:
    case CPU_DEAD_FROZEN:
        migrate_timers((long)hcpu);
        break;
    default:
        break;
    }

    return NOTIFY_OK;
}
```

This chapter will not describe `hotplug` related events in the Linux kernel source code, but if you are interesting in such things, you can find implementation of the `migrate_timers` function in the kernel/time/timer.c source code file.

The last step in the `init_timers` function is the call of the:

```
open_softirq(TIMER_SOFTIRQ, run_timer_softirq);
```

function. The `open_softirq` function may be already familar to you if you have read the ninth part about the interrupts and interrupt handling in the Linux kernel. In short words, the `open_softirq` function defined in the kernel/softirq.c source code file and executes initialization of the deferred interrupt handler.

In our case the deferred function is the `run_timer_softirq` function that is will be called after a hardware interrupt in the `do_IRQ` function which defined in the arch/x86/kernel/irq.c source code file. The main point of this function is to handle a software dynamic timer. The Linux kernel does not do this thing during the hardware timer interrupt handling because this is time consuming operation.

Let's look on the implementation of the `run_timer_softirq` function:

```
static void run_timer_softirq(struct softirq_action *h)
{
    struct tvec_base *base = this_cpu_ptr(&tvec_bases);

    if (time_after_eq(jiffies, base->timer_jiffies))
        __run_timers(base);
}
```

At the beginning of the `run_timer_softirq` function we get a `dynamic` timer for a current processor and compares the current value of the jiffies with the value of the `timer_jiffies` for the current structure by the call of the `time_after_eq` macro which is defined in the include/linux/jiffies.h header file:

```
#define time_after_eq(a,b)          \
    (typecheck(unsigned long, a) && \
     typecheck(unsigned long, b) && \
    ((long)((a) - (b)) >= 0))
```

Reclaim that the `timer_jiffies` field of the `tvec_base` structure represents the relative time when functions delayed by the given timer will be executed. So we compare these two values and if the current time represented by the `jiffies` is greater than `base->timer_jiffies`, we call the `__run_timers` function that defined in the same source code file. Let's look on the implementation of this function.

As I just wrote, the `__run_timers` function runs all expired timers for a given processor. This function starts from the acquiring of the `tvec_base's` lock to protect the `tvec_base` structure

```
static inline void __run_timers(struct tvec_base *base)
{
    struct timer_list *timer;

    spin_lock_irq(&base->lock);
    ...
    ...
    ...
    spin_unlock_irq(&base->lock);
}
```

After this it starts the loop while the `timer_jiffies` will not be greater than the `jiffies`:

```
while (time_after_eq(jiffies, base->timer_jiffies)) {
    ...
    ...
    ...
}
```

We can find many different manipulations in the our loop, but the main point is to find expired timers and call delayed functions. First of all we need to calculate the `index` of the `base->tv1` list that stores the next timer to be handled with the following expression:

```
index = base->timer_jiffies & TVR_MASK;
```

where the `TVR_MASK` is a mask for the getting of the `tvec_root->vec` elements. As we got the index with the next timer which must be handled we check its value. If the index is zero, we go through all lists in our cascade table `tv2`, `tv3` and etc., and rehashing it with the call of the `cascade` function:

```
if (!index &&
    (!cascade(base, &base->tv2, INDEX(0))) &&
        (!cascade(base, &base->tv3, INDEX(1))) &&
                !cascade(base, &base->tv4, INDEX(2)))
        cascade(base, &base->tv5, INDEX(3));
```

After this we increase the value of the `base->timer_jiffies`:

```
++base->timer_jiffies;
```

In the last step we are executing a corresponding function for each timer from the list in a following loop:

```
hlist_move_list(base->tv1.vec + index, head);

while (!hlist_empty(head)) {
    ...
    ...
    ...
    timer = hlist_entry(head->first, struct timer_list, entry);
    fn = timer->function;
    data = timer->data;

    spin_unlock(&base->lock);
    call_timer_fn(timer, fn, data);
    spin_lock(&base->lock);

    ...
    ...
    ...
}
```

where the `call_timer_fn` just call the given function:

```
static void call_timer_fn(struct timer_list *timer, void (*fn)(unsigned long),
                          unsigned long data)
{
    ...
    ...
    ...
    fn(data);
    ...
    ...
    ...
}
```

That's all. The Linux kernel has infrastructure for `dynamic timers` from this moment. We will not dive into this interesting theme. As I already wrote the `timers` is a widely used concept in the Linux kernel and nor one part, nor two parts will not cover understanding of such things how it implemented and how it works. But now we know about this concept, why does the Linux kernel needs in it and some data structures around it.

Now let's look usage of `dynamic timers` in the Linux kernel.

Usage of dynamic timers

As you already can noted, if the Linux kernel provides a concept, it also provides API for managing of this concept and the `dynamic timers` concept is not exception here. To use a timer in the Linux kernel code, we must define a variable with a `timer_list` type. We can initialize our `timer_list` structure in two ways. The first is to use the `init_timer` macro that defined in the include/linux/timer.h header file:

```
#define init_timer(timer)    \
    __init_timer((timer), 0)

#define __init_timer(_timer, _flags)   \
         init_timer_key((_timer), (_flags), NULL, NULL)
```

where the `init_timer_key` function just calls the:

```
do_init_timer(timer, flags, name, key);
```

function which fields the given `timer` with default values. The second way is to use the:

```
#define TIMER_INITIALIZER(_function, _expires, _data)        \
    __TIMER_INITIALIZER((_function), (_expires), (_data), 0)
```

macro which will initilize the given `timer_list` structure too.

After a `dynamic timer` is initialzed we can start this `timer` with the call of the:

```
void add_timer(struct timer_list * timer);
```

function and stop it with the:

```
int del_timer(struct timer_list * timer);
```

function.

That's all.

Conclusion

This is the end of the fourth part of the chapter that describes timers and timer management related stuff in the Linux kernel. In the previous part we got acquainted with the two new concepts: the `tick broadcast` framework and the `NO_HZ` mode. In this part we continued to dive into time managemented related stuff and got acquainted with the new concept - `dynamic timer` or software timer. We didn't saw implementation of a `dynamic timers` management code in details in this part but saw data structures and API around this concept.

In the next part we will continue to dive into timer management related things in the Linux kernel and will see new concept for us - `timers`.

If you have questions or suggestions, feel free to ping me in twitter 0xAX, drop me email or just create issue.

Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-insides.

Links

- IP
- netfilter
- network
- cpumask
- interrupt
- jiffies

- per-cpu
- spinlock
- procfs
- previous part

Timers and time management in the Linux kernel. Part 5.

Introduction to the `clockevents` framework

This is fifth part of the chapter which describes timers and time management related stuff in the Linux kernel. As you might noted from the title of this part, the `clockevents` framework will be discussed. We already saw one framework in the second part of this chapter. It was `clocksource` framework. Both of these frameworks represent timekeeping abstractions in the Linux kernel.

At first let's refresh your memory and try to remember what is it `clocksource` framework and and what its purpose. The main goal of the `clocksource` framework is to provide `timeline`. As described in the documentation:

> For example issuing the command 'date' on a Linux system will eventually read the clock source to determine exactly what time it is.

The Linux kernel supports many different clock sources. You can find some of them in the drivers/closksource. For example old good Intel 8253 - programmable interval timer with `1193182` Hz frequency, yet another one - ACPI PM timer with `3579545` Hz frequency. Besides the drivers/closksource directory, each architecture may provide own architecture-specific clock sources. For example x86 architecture provides High Precision Event Timer, or for example powerpc provides access to the processor timer through `timebase` register.

Each clock source provides monotonic atomic counter. As I already wrote, the Linux kernel supports a huge set of different clock source and each clock source has own parameters like frequency. The main goal of the `clocksource` framework is to provide API to select best available clock source in the system i.e. a clock source with the highest frequency. Additional goal of the `clocksource` framework is to represent an atomic counter provided by a clock source in human units. In this time, nanoseconds are the favorite choice for the time value units of the given clock source in the Linux kernel.

The `clocksource` framework represented by the `clocksource` structure which is defined in the include/linux/clocksource.h header code file which contains `name` of a clock source, rating of certain clock source in the system (a clock source with the higher frequency has the biggest rating in the system), `list` of all registered clock source in the system, `enable` and `disable` fields to enable and disable a clock source, pointer to the `read` function which must return an atomic counter of a clock source and etc.

Additionally the `clocksource` structure provides two fields: `mult` and `shift` which are needed for translation of an atomic counter which is provided by a certain clock source to the human units, i.e. nanoseconds. Translation occurs via following formula:

```
ns ~= (clocksource * mult) >> shift
```

As we already know, besides the `clocksource` structure, the `clocksource` framework provides an API for registration of clock source with different frequency scale factor:

```
static inline int clocksource_register_hz(struct clocksource *cs, u32 hz)
static inline int clocksource_register_khz(struct clocksource *cs, u32 khz)
```

A clock source unregistration:

```
int clocksource_unregister(struct clocksource *cs)
```

and etc.

Additionally to the `clocksource` framework, the Linux kernel provides `clockevents` framework. As described in the documentation:

> Clock events are the conceptual reverse of clock sources

Main goal of the is to manage clock event devices or in other words - to manage devices that allow to register an event or in other words interrupt that is going to happen at a defined point of time in the future.

Now we know a little about the `clockevents` framework in the Linux kernel, and now time is to see on it API.

API of `clockevents` framework

The main structure which described a clock event device is `clock_event_device` structure. This structure is defined in the include/linux/clockchips.h header file and contains a huge set of fields. as well as the `clocksource` structure it has `name` fields which contains human readable name of a clock event device, for example local APIC timer:

```
static struct clock_event_device lapic_clockevent = {
    .name                   = "lapic",
    ...
    ...
    ...
}
```

Addresses of the `event_handler`, `set_next_event`, `next_event` functions for a certain clock event device which are an interrupt handler, setter of next event and local storage for next event respectively. Yet another field of the `clock_event_device` structure is - `features` field. Its value maybe on of the following generic features:

```
#define CLOCK_EVT_FEAT_PERIODIC    0x000001
#define CLOCK_EVT_FEAT_ONESHOT        0x000002
```

Where the `CLOCK_EVT_FEAT_PERIODIC` represents device which may be programmed to generate events periodically. The `CLOCK_EVT_FEAT_ONESHOT` represents device which may generate an event only once. Besides these two features, there are also architecture-specific features. For example x86_64 supports two additional features:

```
#define CLOCK_EVT_FEAT_C3STOP        0x000008
```

The first `CLOCK_EVT_FEAT_C3STOP` means that a clock event device will be stopped in the C3 state. Additionally the `clock_event_device` structure has `mult` and `shift` fields as well as `clocksource` structure. The `clocksource` structure also contains other fields, but we will consider it later.

After we considered part of the `clock_event_device` structure, time is to look at the `API` of the `clockevents` framework. To work with a clock event device, first of all we need to initialize `clock_event_device` structure and register a clock events device. The `clockevents` framework provides following `API` for registration of clock event devices:

```
void clockevents_register_device(struct clock_event_device *dev)
{
   ...
   ...
   ...
}
```

This function defined in the kernel/time/clockevents.c source code file and as we may see, the `clockevents_register_device` function takes only one parameter:

- address of a `clock_event_device` structure which represents a clock event device.

So, to register a clock event device, at first we need to initialize `clock_event_device` structure with parameters of a certain clock event device. Let's take a look at one random clock event device in the Linux kernel source code. We can find one in the drivers/closksource directory or try to take a look at an architecture-specific clock event device. Let's take for example - Periodic Interval Timer (PIT) for at91sam926x. You can find its implementation in the drivers/closksource.

First of all let's look at initialization of the `clock_event_device` structure. This occurs in the `at91sam926x_pit_common_init` function:

```
struct pit_data {
    ...
    ...
    struct clock_event_device       clkevt;
    ...
    ...
};

static void __init at91sam926x_pit_common_init(struct pit_data *data)
{
    ...
    ...
    ...
    data->clkevt.name = "pit";
    data->clkevt.features = CLOCK_EVT_FEAT_PERIODIC;
    data->clkevt.shift = 32;
    data->clkevt.mult = div_sc(pit_rate, NSEC_PER_SEC, data->clkevt.shift);
    data->clkevt.rating = 100;
    data->clkevt.cpumask = cpumask_of(0);

    data->clkevt.set_state_shutdown = pit_clkevt_shutdown;
    data->clkevt.set_state_periodic = pit_clkevt_set_periodic;
    data->clkevt.resume = at91sam926x_pit_resume;
    data->clkevt.suspend = at91sam926x_pit_suspend;
    ...
}
```

Here we can see that `at91sam926x_pit_common_init` takes one parameter - pointer to the `pit_data` structure which contains `clock_event_device` structure which will contain clock event related information of the `at91sam926x` periodic Interval Timer. At the start we fill `name` of the timer device and its `features`. In our case we deal with periodic timer which as we already know may be programmed to generate events periodically.

The next two fields `shift` and `mult` are familiar to us. They will be used to translate counter of our timer to nanoseconds. After this we set rating of the timer to `100`. This means if there will not be timers with higher rating in the system, this timer will be used for

timekeeping. The next field - `cpumask` indicates for which processors in the system the device will work. In our case, the device will work for the first processor. The `cpumask_of` macro defined in the include/linux/cpumask.h header file and just expands to the call of the:

```
#define cpumask_of(cpu) (get_cpu_mask(cpu))
```

Where the `get_cpu_mask` returns the cpumask containing just a given `cpu` number. More about `cpumasks` concept you may read in the CPU masks in the Linux kernel part. In the last four lines of code we set callbacks for the clock event device suspend/resume, device shutdown and update of the clock event device state.

After we finished with the initialization of the `at91sam926x` periodic timer, we can register it by the call of the following functions:

```
clockevents_register_device(&data->clkevt);
```

Now we can consider implementation of the `clockevent_register_device` function. As I already wrote above, this function is defined in the kernel/time/clockevents.c source code file and starts from the initialization of the initial event device state:

```
clockevent_set_state(dev, CLOCK_EVT_STATE_DETACHED);
```

Actually, an event device may be in one of this states:

```
enum clock_event_state {
    CLOCK_EVT_STATE_DETACHED,
    CLOCK_EVT_STATE_SHUTDOWN,
    CLOCK_EVT_STATE_PERIODIC,
    CLOCK_EVT_STATE_ONESHOT,
    CLOCK_EVT_STATE_ONESHOT_STOPPED,
};
```

Where:

- `CLOCK_EVT_STATE_DETACHED` - a clock event device is not not used by `clockevents` framework. Actually it is initial state of all clock event devices;
- `CLOCK_EVT_STATE_SHUTDOWN` - a clock event device is powered-off;
- `CLOCK_EVT_STATE_PERIODIC` - a clock event device may be programmed to generate event periodically;
- `CLOCK_EVT_STATE_ONESHOT` - a clock event device may be programmed to generate event only once;
- `CLOCK_EVT_STATE_ONESHOT_STOPPED` - a clock event device was programmed to generate

event only once and now it is temporary stopped.

The implementation of the `clock_event_set_state` function is pretty easy:

```
static inline void clockevent_set_state(struct clock_event_device *dev,
                                        enum clock_event_state state)
{
    dev->state_use_accessors = state;
}
```

As we can see, it just fills the `state_use_accessors` field of the given `clock_event_device` structure with the given value which is in our case is `CLOCK_EVT_STATE_DETACHED`. Actually all clock event devices has this initial state during registration. The `state_use_accessors` field of the `clock_event_device` structure provides `current` state of the clock event device.

After we have set initial state of the given `clock_event_device` structure we check that the `cpumask` of the given clock event device is not zero:

```
if (!dev->cpumask) {
    WARN_ON(num_possible_cpus() > 1);
    dev->cpumask = cpumask_of(smp_processor_id());
}
```

Remember that we have set the `cpumask` of the `at91sam926x` periodic timer to first processor. If the `cpumask` field is zero, we check the number of possible processors in the system and print warning message if it is less than on. Additionally we set the `cpumask` of the given clock event device to the current processor. If you are interested in how the `smp_processor_id` macro is implemented, you can read more about it in the fourth part of the Linux kernel initialization process chapter.

After this check we lock the actual code of the clock event device registration by the call following macros:

```
raw_spin_lock_irqsave(&clockevents_lock, flags);
...
...
...
raw_spin_unlock_irqrestore(&clockevents_lock, flags);
```

Additionally the `raw_spin_lock_irqsave` and the `raw_spin_unlock_irqrestore` macros disable local interrupts, however interrupts on other processors still may occur. We need to do it to prevent potential deadlock if we adding new clock event device to the list of clock event devices and an interrupt occurs from other clock event device.

We can see following code of clock event device registration between the `raw_spin_lock_irqsave` and `raw_spin_unlock_irqrestore` macros:

```
list_add(&dev->list, &clockevent_devices);
tick_check_new_device(dev);
clockevents_notify_released();
```

First of all we add the given clock event device to the list of clock event devices which is represented by the `clockevent_devices` :

```
static LIST_HEAD(clockevent_devices);
```

At the next step we call the `tick_check_new_device` function which is defined in the kernel/time/tick-common.c source code file and checks do the new registered clock event device should be used or not. The `tick_check_new_device` function checks the given `clock_event_device` gets the current registered tick device which is represented by the `tick_device` structure and compares their ratings and features. Actually `CLOCK_EVT_STATE_ONESHOT` is preferred:

```
static bool tick_check_preferred(struct clock_event_device *curdev,
                                 struct clock_event_device *newdev)
{
    if (!(newdev->features & CLOCK_EVT_FEAT_ONESHOT)) {
        if (curdev && (curdev->features & CLOCK_EVT_FEAT_ONESHOT))
            return false;
        if (tick_oneshot_mode_active())
            return false;
    }

    return !curdev ||
        newdev->rating > curdev->rating ||
           !cpumask_equal(curdev->cpumask, newdev->cpumask);
}
```

If the new registered clock event device is more preferred than old tick device, we exchange old and new registered devices and install new device:

```
clockevents_exchange_device(curdev, newdev);
tick_setup_device(td, newdev, cpu, cpumask_of(cpu));
```

The `clockevents_exchange_device` function releases or in other words deleted the old clock event device from the `clockevent_devices` list. The next function - `tick_setup_device` as we may understand from its name, setups new tick device. This function check the mode of the

new registered clock event device and call the `tick_setup_periodic` function or the `tick_setup_oneshot` depends on the tick device mode:

```
if (td->mode == TICKDEV_MODE_PERIODIC)
    tick_setup_periodic(newdev, 0);
else
    tick_setup_oneshot(newdev, handler, next_event);
```

Both of this functions calls the `clockevents_switch_state` to change state of the clock event device and the `clockevents_program_event` function to set next event of clock event device based on delta between the maximum and minimum difference current time and time for the next event. The `tick_setup_periodic` :

```
clockevents_switch_state(dev, CLOCK_EVT_STATE_PERIODIC);
clockevents_program_event(dev, next, false))
```

and the `tick_setup_oneshot_periodic` :

```
clockevents_switch_state(newdev, CLOCK_EVT_STATE_ONESHOT);
clockevents_program_event(newdev, next_event, true);
```

The `clockevents_switch_state` function checks that the clock event device is not in the given state and calls the `__clockevents_switch_state` function from the same source code file:

```
if (clockevent_get_state(dev) != state) {
    if (__clockevents_switch_state(dev, state))
        return;
```

The `__clockevents_switch_state` function just makes a call of the certain callback depends on the given state:

```
static int __clockevents_switch_state(struct clock_event_device *dev,
                      enum clock_event_state state)
{
    if (dev->features & CLOCK_EVT_FEAT_DUMMY)
        return 0;

    switch (state) {
    case CLOCK_EVT_STATE_DETACHED:
    case CLOCK_EVT_STATE_SHUTDOWN:
        if (dev->set_state_shutdown)
            return dev->set_state_shutdown(dev);
        return 0;

    case CLOCK_EVT_STATE_PERIODIC:
        if (!(dev->features & CLOCK_EVT_FEAT_PERIODIC))
            return -ENOSYS;
        if (dev->set_state_periodic)
            return dev->set_state_periodic(dev);
        return 0;
    ...
    ...
    ...
```

In our case for `at91sam926x` periodic timer, the state is the `CLOCK_EVT_FEAT_PERIODIC` :

```
data->clkevt.features = CLOCK_EVT_FEAT_PERIODIC;
data->clkevt.set_state_periodic = pit_clkevt_set_periodic;
```

So, for the `pit_clkevt_set_periodic` callback will be called. If we will read the documentation of the Periodic Interval Timer (PIT) for at91sam926x, we will see that there is `Periodic Interval Timer Mode Register` which allows us to control of periodic interval timer.

It looks like:

```
31                                                  25         24
+------------------------------------------------------------------+
|                                          |  PITIEN  |  PITEN   |
+------------------------------------------------------------------+
23                             19                               16
+------------------------------------------------------------------+
|                              |              PIV                 |
+------------------------------------------------------------------+
15                                                               8
+------------------------------------------------------------------+
|                             PIV                                 |
+------------------------------------------------------------------+
7                                                                0
+------------------------------------------------------------------+
|                             PIV                                 |
+------------------------------------------------------------------+
```

Where `PIV` or `Periodic Interval Value` - defines the value compared with the primary `20-bit` counter of the Periodic Interval Timer. The `PITEN` or `Period Interval Timer Enabled` if the bit is `1` and the `PITIEN` or `Periodic Interval Timer Interrupt Enable` if the bit is `1`. So, to set periodic mode, we need to set `24`, `25` bits in the `Periodic Interval Timer Mode Register`. And we are doing it in the `pit_clkevt_set_periodic` function:

```
static int pit_clkevt_set_periodic(struct clock_event_device *dev)
{
        struct pit_data *data = clkevt_to_pit_data(dev);
        ...
        ...
        ...
        pit_write(data->base, AT91_PIT_MR,
                  (data->cycle - 1) | AT91_PIT_PITEN | AT91_PIT_PITIEN);

        return 0;
}
```

Where the `AT91_PT_MR`, `AT91_PT_PITEN` and the `AT91_PIT_PITIEN` are declared as:

```
#define AT91_PIT_MR             0x00
#define AT91_PIT_PITIEN        BIT(25)
#define AT91_PIT_PITEN         BIT(24)
```

After the setup of the new clock event device is finished, we can return to the `clockevents_register_device` function. The last function in the `clockevents_register_device` function is:

```
clockevents_notify_released();
```

This function checks the `clockevents_released` list which contains released clock event devices (remember that they may occur after the call of the `clockevents_exchange_device` function). If this list is not empty, we go through clock event devices from the `clock_events_released` list and delete it from the `clockevent_devices` :

```
static void clockevents_notify_released(void)
{
    struct clock_event_device *dev;

    while (!list_empty(&clockevents_released)) {
        dev = list_entry(clockevents_released.next,
                 struct clock_event_device, list);
        list_del(&dev->list);
        list_add(&dev->list, &clockevent_devices);
        tick_check_new_device(dev);
    }
}
```

That's all. From this moment we have registered new clock event device. So the usage of the `clockevents` framework is simple and clear. Architectures registered their clock event devices, in the clock events core. Users of the clockevents core can get clock event devices for their use. The `clockevents` framework provides notification mechanisms for various clock related management events like a clock event device registered or unregistered, a processor is offlined in system which supports CPU hotplug and etc.

We saw implementation only of the `clockevents_register_device` function. But generally, the clock event layer API is small. Besides the `API` for clock event device registration, the `clockevents` framework provides functions to schedule the next event interrupt, clock event device notification service and support for suspend and resume for clock event devices.

If you want to know more about `clockevents` API you can start to research following source code and header files: kernel/time/tick-common.c, kernel/time/clockevents.c and include/linux/clockchips.h.

That's all.

Conclusion

This is the end of the fifth part of the chapter that describes timers and timer management related stuff in the Linux kernel. In the previous part got acquainted with the `timers` concept. In this part we continued to learn time management related stuff in the Linux kernel

and saw a little about yet another framework - `clockevents` .

If you have questions or suggestions, feel free to ping me in twitter 0xAX, drop me email or just create issue.

Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-insides.

Links

- timekeeping documentation
- Intel 8253
- programmable interval timer
- ACPI pdf
- x86
- High Precision Event Timer
- powerpc
- frequency
- API
- nanoseconds
- interrupt
- interrupt handler
- local APIC
- C3 state
- Periodic Interval Timer (PIT) for at91sam926x
- CPU masks in the Linux kernel
- deadlock
- CPU hotplug
- previous part

Timers and time management in the Linux kernel. Part 6.

x86_64 related clock sources

This is sixth part of the chapter which describes timers and time management related stuff in the Linux kernel. In the previous part we saw `clockevents` framework and now we will continue to dive into time management related stuff in the Linux kernel. This part will describe implementation of x86 architecture related clock sources (more about `clocksource` concept you can read in the second part of this chapter).

First of all we must know what clock sources may be used at `x86` architecture. It is easy to know from the sysfs or from content of the `/sys/devices/system/clocksource/clocksource0/available_clocksource`. The `/sys/devices/system/clocksource/clocksourceN` provides two special files to achieve this:

* `available_clocksource` - provides information about available clock sources in the system;
* `current_clocksource` - provides information about currently used clock source in the system.

So, let's look:

```
$ cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
```

We can see that there are three registered clock sources in my system:

* `tsc` - Time Stamp Counter;
* `hpet` - High Precision Event Timer;
* `acpi_pm` - ACPI Power Management Timer.

Now let's look at the second file which provides best clock source (a clock source which has the best rating in the system):

```
$ cat /sys/devices/system/clocksource/clocksource0/current_clocksource
tsc
```

For me it is Time Stamp Counter. As we may know from the second part of this chapter, which describes internals of the `clocksource` framework in the Linux kernel, the best clock source in a system is a clock source with the best (highest) rating or in other words with the highest frequency.

Frequency of the ACPI power management timer is `3.579545 MHz`. Frequency of the High Precision Event Timer is at least `10 MHz`. And the frequency of the Time Stamp Counter depends on processor. For example On older processors, the `Time Stamp Counter` was counting internal processor clock cycles. This means its frequency changed when the processor's frequency scaling changed. The situation has changed for newer processors. Newer processors have an `invariant Time Stamp counter` that increments at a constant rate in all operational states of processor. Actually we can get its frequency in the output of the `/proc/cpuinfo`. For example for the first processor in the system:

```
$ cat /proc/cpuinfo
...
model name    : Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz
...
```

And although Intel manual says that the frequency of the `Time Stamp Counter`, while constant, is not necessarily the maximum qualified frequency of the processor, or the frequency given in the brand string, anyway we may see that it will be much more than frequency of the `ACPI PM` timer or `High Precision Event Timer`. And we can see that the clock source with the best rating or highest frequency is current in the system.

You can note that besides these three clock source, we don't see yet another two familiar us clock sources in the output of the `/sys/devices/system/clocksource/clocksource0/available_clocksource`. These clock sources are `jiffy` and `refined_jiffies`. We don't see them because this filed maps only high resolution clock sources or in other words clock sources with the CLOCK_SOURCE_VALID_FOR_HRES flag.

As I already wrote above, we will consider all of these three clock sources in this part. We will consider it in order of their initialization or:

- `hpet`;
- `acpi_pm`;
- `tsc`.

We can make sure that the order is exactly like this in the output of the dmesg util:

```
$ dmesg | grep clocksource
[    0.000000] clocksource: refined-jiffies: mask: 0xffffffff max_cycles: 0xffffffff,
max_idle_ns: 1910969940391419 ns
[    0.000000] clocksource: hpet: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns
: 133484882848 ns
[    0.094369] clocksource: jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max_idle
_ns: 1911260446275000 ns
[    0.186498] clocksource: Switched to clocksource hpet
[    0.196827] clocksource: acpi_pm: mask: 0xffffff max_cycles: 0xffffff, max_idle_ns:
 2085701024 ns
[    1.413685] tsc: Refined TSC clocksource calibration: 3999.981 MHz
[    1.413688] clocksource: tsc: mask: 0xffffffffffffffff max_cycles: 0x73509721780, m
ax_idle_ns: 881591102108 ns
[    2.413748] clocksource: Switched to clocksource tsc
```

The first clock source is the High Precision Event Timer, so let's start from it.

High Precision Event Timer

The implementation of the `High Precision Event Timer` for the x86 architecture is located in the arch/x86/kernel/hpet.c source code file. Its initialization starts from the call of the `hpet_enable` function. This function is called during Linux kernel initialization. If we will look into `start_kernel` function from the init/main.c source code file, we will see that after the all architecture-specific stuff initialized, early console is disabled and time management subsystem already ready, call of the following function:

```
if (late_time_init)
    late_time_init();
```

which does initialization of the late architecture specific timers after early jiffy counter already initialized. The definition of the `late_time_init` function for the `x86` architecture is located in the arch/x86/kernel/time.c source code file. It looks pretty easy:

```
static __init void x86_late_time_init(void)
{
    x86_init.timers.timer_init();
    tsc_init();
}
```

As we may see, it does initialization of the `x86` related timer and initialization of the `Time Stamp Counter`. The seconds we will see in the next paragraph, but now let's consider the call of the `x86_init.timers.timer_init` function. The `timer_init` points to the

`hpet_time_init` function from the same source code file. We can verify this by looking on the definition of the `x86_init` structure from the arch/x86/kernel/x86_init.c:

```
struct x86_init_ops x86_init __initdata = {
   ...
   ...
   ...
   .timers = {
        .setup_percpu_clockev    = setup_boot_APIC_clock,
        .timer_init        = hpet_time_init,
        .wallclock_init        = x86_init_noop,
   },
   ...
   ...
   ...
```

The `hpet_time_init` function does setup of the programmable interval timer if we can not enable `High Precision Event Timer` and setups default timer IRQ for the enabled timer:

```
void __init hpet_time_init(void)
{
    if (!hpet_enable())
        setup_pit_timer();
    setup_default_timer_irq();
}
```

First of all the `hpet_enable` function check we can enable `High Precision Event Timer` in the system by the call of the `is_hpet_capable` function and if we can, we map a virtual address space for it:

```
int __init hpet_enable(void)
{
    if (!is_hpet_capable())
        return 0;

    hpet_set_mapping();
}
```

The `is_hpet_capable` function checks that we didn't pass `hpet=disable` to the kernel command line and the `hpet_address` is received from the ACPI HPET table. The `hpet_set_mapping` function just maps the virtual address spaces for the timer registers:

```
hpet_virt_address = ioremap_nocache(hpet_address, HPET_MMAP_SIZE);
```

As we can read in the IA-PC HPET (High Precision Event Timers) Specification:

> The timer register space is 1024 bytes

So, the `HPET_MMAP_SIZE` is `1024` bytes too:

```
#define HPET_MMAP_SIZE        1024
```

After we mapped virtual space for the `High Precision Event Timer`, we read `HPET_ID` register to get number of the timers:

```
id = hpet_readl(HPET_ID);

last = (id & HPET_ID_NUMBER) >> HPET_ID_NUMBER_SHIFT;
```

We need to get this number to allocate correct amount of space for the `General Configuration Register` of the `High Precision Event Timer`:

```
cfg = hpet_readl(HPET_CFG);

hpet_boot_cfg = kmalloc((last + 2) * sizeof(*hpet_boot_cfg), GFP_KERNEL);
```

After the space is allocated for the configuration register of the `High Precision Event Timer`, we allow to main counter to run, and allow timer interrupts if they are enabled by the setting of `HPET_CFG_ENABLE` bit in the configuration register for all timers. In the end we just register new clock source by the call of the `hpet_clocksource_register` function:

```
if (hpet_clocksource_register())
    goto out_nohpet;
```

which just calls already familiar

```
clocksource_register_hz(&clocksource_hpet, (u32)hpet_freq);
```

function. Where the `clocksource_hpet` is the `clocksource` structure with the rating `250` (remember rating of the previous `refined_jiffies` clock source was `2`), name - `hpet` and `read_hpet` callback for the reading of atomic counter provided by the `High Precision Event Timer`:

```
static struct clocksource clocksource_hpet = {
    .name         = "hpet",
    .rating         = 250,
    .read         = read_hpet,
    .mask         = HPET_MASK,
    .flags         = CLOCK_SOURCE_IS_CONTINUOUS,
    .resume         = hpet_resume_counter,
    .archdata     = { .vclock_mode = VCLOCK_HPET },
};
```

After the `clocksource_hpet` is registered, we can return to the `hpet_time_init()` function from the arch/x86/kernel/time.c source code file. We can remember that the last step is the call of the:

```
setup_default_timer_irq();
```

function in the `hpet_time_init()` . The `setup_default_timer_irq` function checks existence of `legacy` IRQs or in other words support for the i8259 and setups IRQ0 depends on this.

That's all. From this moment the High Precision Event Timer clock source registered in the Linux kernel `clock source` framework and may be used from generic kernel code via the `read_hpet` :

```
static cycle_t read_hpet(struct clocksource *cs)
{
    return (cycle_t)hpet_readl(HPET_COUNTER);
}
```

function which just reads and returns atomic counter from the `Main Counter Register` .

ACPI PM timer

The seconds clock source is ACPI Power Management Timer. Implementation of this clock source is located in the drivers/clocksource/acpi_pm.c source code file and starts from the call of the `init_acpi_pm_clocksource` function during `fs` initcall.

If we will look at implementation of the `init_acpi_pm_clocksource` function, we will see that it starts from the check of the value of `pmtmr_ioport` variable:

```
static int __init init_acpi_pm_clocksource(void)
{
    ...
    ...
    ...
    if (!pmtmr_ioport)
        return -ENODEV;
    ...
    ...
    ...
```

This `pmtmr_ioport` variable contains extended address of the `Power Management Timer Control Register Block`. It gets its value in the `acpi_parse_fadt` function which is defined in the arch/x86/kernel/acpi/boot.c source code file. This function parses `FADT` or `Fixed ACPI Description Table` ACPI table and tries to get the values of the `X_PM_TMR_BLK` field which contains extended address of the `Power Management Timer Control Register Block`, represented in `Generic Address Structure` format:

```
static int __init acpi_parse_fadt(struct acpi_table_header *table)
{
#ifdef CONFIG_X86_PM_TIMER
        ...
        ...
        ...
        pmtmr_ioport = acpi_gbl_FADT.xpm_timer_block.address;
        ...
        ...
        ...
#endif
    return 0;
}
```

So, if the `CONFIG_X86_PM_TIMER` Linux kernel configuration option is disabled or something going wrong in the `acpi_parse_fadt` function, we can't access the `Power Management Timer` register and return from the `init_acpi_pm_clocksource`. In other way, if the value of the `pmtmr_ioport` variable is not zero, we check rate of this timer and register this clock source by the call of the:

```
clocksource_register_hz(&clocksource_acpi_pm, PMTMR_TICKS_PER_SEC);
```

function. After the call of the `clocksource_register_hs`, the `acpi_pm` clock source will be registered in the `clocksource` framework of the Linux kernel:

```
static struct clocksource clocksource_acpi_pm = {
    .name         = "acpi_pm",
    .rating         = 200,
    .read         = acpi_pm_read,
    .mask         = (cycle_t)ACPI_PM_MASK,
    .flags         = CLOCK_SOURCE_IS_CONTINUOUS,
};
```

with the rating - `200` and the `acpi_pm_read` callback to read atomic counter provided by the `acpi_pm` clock source. The `acpi_pm_read` function just executes `read_pmtmr` function:

```
static cycle_t acpi_pm_read(struct clocksource *cs)
{
    return (cycle_t)read_pmtmr();
}
```

which reads value of the `Power Management Timer` register. This register has following structure:

```
+-------------------------------+----------------------------------+
|                               |                                  |
|  upper eight bits of a        |      running count of the        |
| 32-bit power management timer |     power management timer       |
|                               |                                  |
+-------------------------------+----------------------------------+
31          E_TMR_VAL           24               TMR_VAL           0
```

Address of this register is stored in the `Fixed ACPI Description Table` ACPI table and we already have it in the `pmtmr_ioport`. So, the implementation of the `read_pmtmr` function is pretty easy:

```
static inline u32 read_pmtmr(void)
{
    return inl(pmtmr_ioport) & ACPI_PM_MASK;
}
```

We just read the value of the `Power Management Timer` register and mask its `24` bits.

That's all. Now we move to the last clock source in this part - `Time Stamp Counter`.

Time Stamp Counter

The third and last clock source in this part is - Time Stamp Counter clock source and its implementation is located in the arch/x86/kernel/tsc.c source code file. We already saw the `x86_late_time_init` function in this part and initialization of the Time Stamp Counter starts from this place. This function calls the `tsc_init()` function from the arch/x86/kernel/tsc.c source code file.

At the beginning of the `tsc_init` function we can see check, which checks that a processor has support of the `Time Stamp Counter`:

```
void __init tsc_init(void)
{
    u64 lpj;
    int cpu;

    if (!cpu_has_tsc) {
        setup_clear_cpu_cap(X86_FEATURE_TSC_DEADLINE_TIMER);
        return;
    }
    ...
    ...
    ...
```

The `cpu_has_tsc` macro expands to the call of the `cpu_has` macro:

```
#define cpu_has_tsc        boot_cpu_has(X86_FEATURE_TSC)

#define boot_cpu_has(bit)    cpu_has(&boot_cpu_data, bit)

#define cpu_has(c, bit)                             \
    (__builtin_constant_p(bit) && REQUIRED_MASK_BIT_SET(bit) ? 1 :    \
     test_cpu_cap(c, bit))
```

which check the given bit (the `X86_FEATURE_TSC_DEADLINE_TIMER` in our case) in the `boot_cpu_data` array which is filled during early Linux kernel initialization. If the processor has support of the `Time Stamp Counter`, we get the frequency of the `Time Stamp Counter` by the call of the `calibrate_tsc` function from the same source code file which tries to get frequency from the different source like Model Specific Register, calibrate over programmable interval timer and etc, after this we initialize frequency and scale factor for the all processors in the system:

```
tsc_khz = x86_platform.calibrate_tsc();
cpu_khz = tsc_khz;

for_each_possible_cpu(cpu) {
    cyc2ns_init(cpu);
    set_cyc2ns_scale(cpu_khz, cpu);
}
```

because only first bootstrap processor will call the `tsc_init`. After this we check hat `Time Stamp Counter` is not disabled:

```
if (tsc_disabled > 0)
    return;
...
...
...
check_system_tsc_reliable();
```

and call the `check_system_tsc_reliable` function which sets the `tsc_clocksource_reliable` if bootstrap processor has the `X86_FEATURE_TSC_RELIABLE` feature. Note that we went through the `tsc_init` function, but did not register our clock source. Actual registration of the `Time Stamp Counter` clock source occurs in the:

```
static int __init init_tsc_clocksource(void)
{
    if (!cpu_has_tsc || tsc_disabled > 0 || !tsc_khz)
        return 0;
    ...
    ...
    ...
    if (boot_cpu_has(X86_FEATURE_TSC_RELIABLE)) {
        clocksource_register_khz(&clocksource_tsc, tsc_khz);
        return 0;
    }
```

function. This function called during the `device` initcall. We do it to be sure that the `Time Stamp Counter` clock source will be registered after the High Precision Event Timer clock source.

After these all three clock sources will be registered in the `clocksource` framework and the `Time Stamp Counter` clock source will be selected as active, because it has the highest rating among other clock sources:

```
static struct clocksource clocksource_tsc = {
    .name                   = "tsc",
    .rating                 = 300,
    .read                   = read_tsc,
    .mask                   = CLOCKSOURCE_MASK(64),
    .flags                  = CLOCK_SOURCE_IS_CONTINUOUS | CLOCK_SOURCE_MUST_VERIFY,
    .archdata               = { .vclock_mode = VCLOCK_TSC },
};
```

That's all.

Conclusion

This is the end of the sixth part of the chapter that describes timers and timer management related stuff in the Linux kernel. In the previous part got acquainted with the `clockevents` framework. In this part we continued to learn time management related stuff in the Linux kernel and saw a little about three different clock sources which are used in the x86 architecture. The next part will be last part of this chapter and we will see some user space related stuff, i.e. how some time related system calls implemented in the Linux kernel.

If you have questions or suggestions, feel free to ping me in twitter 0xAX, drop me email or just create issue.

Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-insides.

Links

- x86
- sysfs
- Time Stamp Counter
- High Precision Event Timer
- ACPI Power Management Timer (PDF)
- frequency.
- dmesg
- programmable interval timer
- IRQ
- IA-PC HPET (High Precision Event Timers) Specification
- IRQ0
- i8259
- initcall

* previous part

Timers and time management in the Linux kernel. Part 7.

Time related system calls in the Linux kernel

This is the seventh and last part chapter which describes timers and time management related stuff in the Linux kernel. In the previous part we saw some x86_64 like High Precision Event Timer and Time Stamp Counter. Internal time management is interesting part of the Linux kernel, but of course not only the kernel needs in the `time` concept. Our programs need to know time too. In this part, we will consider implementation of some time management related system calls. These system calls are:

- `clock_gettime` ;
- `gettimeofday` ;
- `nanosleep` .

We will start from simple userspace C program and see all way from the call of the standard library function to the implementation of certain system call. As each architecture provides its own implementation of certain system call, we will consider only x86_64 specific implementations of system calls, as this book is related to this architecture.

Additionally we will not consider concept of system calls in this part, but only implementations of these three system calls in the Linux kernel. If you are interested in what is it a `system call` , there is special chapter about this.

So, let's from the `gettimeofday` system call.

Implementation of the `gettimeofday` system call

As we can understand from the name of the `gettimeofday` , this function returns current time. First of all, let's look on the following simple example:

```
#include <time.h>
#include <sys/time.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    char buffer[40];
    struct timeval time;

    gettimeofday(&time, NULL);

    strftime(buffer, 40, "Current date/time: %m-%d-%Y/%T", localtime(&time.tv_sec));
    printf("%s\n",buffer);

    return 0;
}
```

As you can see, here we call the `gettimeofday` function which takes two parameters: pointer to the `timeval` structure which represents an elapsed tim:

```
struct timeval {
    time_t      tv_sec;     /* seconds */
    suseconds_t tv_usec;    /* microseconds */
};
```

The second parameter of the `gettimeofday` function is pointer to the `timezone` structure which represents a timezone. In our example, we pass address of the `timeval time` to the `gettimeofday` function, the Linux kernel fills the given `timeval` structure and returns it back to us. Additionally, we format the time with the `strftime` function to get something more human readable than elapsed microseconds. Let's see on result:

```
~$ gcc date.c -o date
~$ ./date
Current date/time: 03-26-2016/16:42:02
```

As you already may know, an userspace application does not call a system call directly from the kernel space. Before the actual system call entry will be called, we call a function from the standard library. In my case it is glibc, so I will consider this case. The implementation of the `gettimeofday` function is located in the sysdeps/unix/sysv/linux/x86/gettimeofday.c source code file. As you already may know, the `gettimeofday` is not usual system call. It is located in the special area which is called `vDSO` (you can read more about it in the part which describes this concept).

The `glibc` implementation of the `gettimeofday` tries to resolve the given symbol, in our case this symbol is `__vdso_gettimeofday` by the call of the `_dl_vdso_vsym` internal function. If the symbol will not be resolved, it returns `NULL` and we fallback to the call of the usual system call:

```
return (_dl_vdso_vsym ("__vdso_gettimeofday", &linux26)
  ?: (void*) (&__gettimeofday_syscall));
```

The `gettimeofday` entry is located in the arch/x86/entry/vdso/vclock_gettime.c source code file. As we can see the `gettimeofday` is weak alias of the `__vdso_gettimeofday` :

```
int gettimeofday(struct timeval *, struct timezone *)
    __attribute__((weak, alias("__vdso_gettimeofday")));
```

The `__vdso_gettimeofday` is defined in the same source code file and calls the `do_realtime` function if the given `timeval` is not null:

```
notrace int __vdso_gettimeofday(struct timeval *tv, struct timezone *tz)
{
    if (likely(tv != NULL)) {
        if (unlikely(do_realtime((struct timespec *)tv) == VCLOCK_NONE))
            return vdso_fallback_gtod(tv, tz);
        tv->tv_usec /= 1000;
    }
    if (unlikely(tz != NULL)) {
        tz->tz_minuteswest = gtod->tz_minuteswest;
        tz->tz_dsttime = gtod->tz_dsttime;
    }

    return 0;
}
```

If the `do_realtime` will fail, we fallback to the real system call via call the `syscall` instruction and passing the `__NR_gettimeofday` system call number and the given `timeval` and `timezone` :

```
notrace static long vdso_fallback_gtod(struct timeval *tv, struct timezone *tz)
{
    long ret;

    asm("syscall" : "=a" (ret) :
        "0" (__NR_gettimeofday), "D" (tv), "S" (tz) : "memory");
    return ret;
}
```

The `do_realtime` function gets the time data from the `vsyscall_gtod_data` structure which is defined in the arch/x86/include/asm/vgtod.h header file and contains mapping of the `timespec` structure and a couple of fields which are related to the current clock source in the system. This function fills the given `timeval` structure with values from the `vsyscall_gtod_data` which contains a time related data which is updated via timer interrupt.

First of all we try to access the `gtod` or `global time of day` the `vsyscall_gtod_data` structure via the call of the `gtod_read_begin` and will continue to do it until it will be successful:

```
do {
    seq = gtod_read_begin(gtod);
    mode = gtod->vclock_mode;
    ts->tv_sec = gtod->wall_time_sec;
    ns = gtod->wall_time_snsec;
    ns += vgetsns(&mode);
    ns >>= gtod->shift;
} while (unlikely(gtod_read_retry(gtod, seq)));

ts->tv_sec += __iter_div_u64_rem(ns, NSEC_PER_SEC, &ns);
ts->tv_nsec = ns;
```

As we got access to the `gtod` , we fill the `ts->tv_sec` with the `gtod->wall_time_sec` which stores current time in seconds gotten from the real time clock during initialization of the timekeeping subsystem in the Linux kernel and the same value but in nanoseconds. In the end of this code we just fill the given `timespec` structure with the resulted values.

That's all about the `gettimeofday` system call. The next system call in our list is the `clock_gettime` .

Implementation of the clock_gettime system call

The `clock_gettime` function gets the time which is specified by the second parameter. Generally the `clock_gettime` function takes two parameters:

- `clk_id` - clock identifier;
- `timespec` - address of the `timespec` structure which represent elapsed time.

Let's look on the following simple example:

```
#include <time.h>
#include <sys/time.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    struct timespec elapsed_from_boot;

    clock_gettime(CLOCK_BOOTTIME, &elapsed_from_boot);

    printf("%d - seconds elapsed from boot\n", elapsed_from_boot.tv_sec);

    return 0;
}
```

which prints `uptime` information:

```
~$ gcc uptime.c -o uptime
~$ ./uptime
14180 - seconds elapsed from boot
```

We can easily check the result with the help of the uptime util:

```
~$ uptime
up  3:56
```

The `elapsed_from_boot.tv_sec` represents elapsed time in seconds, so:

```
>>> 14180 / 60
236
>>> 14180 / 60 / 60
3
>>> 14180 / 60 % 60
56
```

The `clock_id` maybe one of the following:

- `CLOCK_REALTIME` - system wide clock which measures real or wall-clock time;
- `CLOCK_REALTIME_COARSE` - faster version of the `CLOCK_REALTIME` ;
- `CLOCK_MONOTONIC` - represents monotonic time since some unspecified starting point;
- `CLOCK_MONOTONIC_COARSE` - faster version of the `CLOCK_MONOTONIC` ;
- `CLOCK_MONOTONIC_RAW` - the same as the `CLOCK_MONOTONIC` but provides non NTP adjusted time.
- `CLOCK_BOOTTIME` - the same as the `CLOCK_MONOTONIC` but plus time that the system was suspended;

- `CLOCK_PROCESS_CPUTIME_ID` - per-process time consumed by all threads in the process;
- `CLOCK_THREAD_CPUTIME_ID` - thread-specific clock.

The `clock_gettime` is not usual syscall too, but as the `gettimeofday`, this system call is placed in the `vDSO` area. Entry of this system call is located in the same source code file - arch/x86/entry/vdso/vclock_gettime.c) as for `gettimeofday`.

The Implementation of the `clock_gettime` depends on the clock id. If we have passed the `CLOCK_REALTIME` clock id, the `do_realtime` function will be called:

```
notrace int __vdso_clock_gettime(clockid_t clock, struct timespec *ts)
{
    switch (clock) {
    case CLOCK_REALTIME:
        if (do_realtime(ts) == VCLOCK_NONE)
            goto fallback;
        break;
    ...
    ...
    ...
fallback:
    return vdso_fallback_gettime(clock, ts);
}
```

In other cases, the `do_{name_of_clock_id}` function is called. Implementations of some of them is similar. For example if we will pass the `CLOCK_MONOTONIC` clock id:

```
...
...
...
case CLOCK_MONOTONIC:
    if (do_monotonic(ts) == VCLOCK_NONE)
        goto fallback;
    break;
...
...
...
```

the `do_monotonic` function will be called which is very similar on the implementation of the `do_realtime`:

```
notrace static int __always_inline do_monotonic(struct timespec *ts)
{
    do {
        seq = gtod_read_begin(gtod);
        mode = gtod->vclock_mode;
        ts->tv_sec = gtod->monotonic_time_sec;
        ns = gtod->monotonic_time_snsec;
        ns += vgetsns(&mode);
        ns >>= gtod->shift;
    } while (unlikely(gtod_read_retry(gtod, seq)));

    ts->tv_sec += __iter_div_u64_rem(ns, NSEC_PER_SEC, &ns);
    ts->tv_nsec = ns;

    return mode;
}
```

We already saw a little about the implementation of this function in the previous paragraph about the `gettimeofday` . There is only one difference here, that the `sec` and `nsec` of our `timespec` value will be based on the `gtod->monotonic_time_sec` instead of `gtod->wall_time_sec` which maps the value of the `tk->tkr_mono.xtime_nsec` or number of nanoseconds elapsed.

That's all.

Implementation of the `nanosleep` system call

The last system call in our list is the `nanosleep` . As you can understand from its name, this function provides `sleeping` ability. Let's look on the following simple example:

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>

int main (void)
{
   struct timespec ts = {5,0};

   printf("sleep five seconds\n");
   nanosleep(&ts, NULL);
   printf("end of sleep\n");

   return 0;
}
```

If we will compile and run it, we will see the first line

```
~$ gcc sleep_test.c -o sleep
~$ ./sleep
sleep five seconds
end of sleep
```

and the second line after five seconds.

The `nanosleep` is not located in the `vDSO` area like the `gettimeofday` and the `clock_gettime` functions. So, let's look how the `real` system call which is located in the kernel space will be called by the standard library. The implementation of the `nanosleep` system call will be called with the help of the syscall instruction. Before the execution of the `syscall` instruction, parameters of the system call must be put in processor registers according to order which is described in the System V Application Binary Interface or in other words:

- `rdi` - first parameter;
- `rsi` - second parameter;
- `rdx` - third parameter;
- `r10` - fourth parameter;
- `r8` - fifth parameter;
- `r9` - sixth parameter.

The `nanosleep` system call has two parameters - two pointers to the `timespec` structures. The system call suspends the calling thread until the given timeout has elapsed. Additionally it will finish if a signal interrupts its execution. It takes two parameters, the first is `timespec` which represents timeout for the sleep. The second parameter is the pointer to the `timespec` structure too and it contains remainder of time if the call of the `nanosleep` was interrupted.

As `nanosleep` has two parameters:

```
int nanosleep(const struct timespec *req, struct timespec *rem);
```

To call system call, we need put the `req` to the `rdi` register, and the `rem` parameter to the `rsi` register. The glibc does these job in the `INTERNAL_SYSCALL` macro which is located in the sysdeps/unix/sysv/linux/x86_64/sysdep.h header file.

```
# define INTERNAL_SYSCALL(name, err, nr, args...) \
  INTERNAL_SYSCALL_NCS (__NR_##name, err, nr, ##args)
```

which takes the name of the system call, storage for possible error during execution of system call, number of the system call (all `x86_64` system calls you can find in the system calls table) and arguments of certain system call. The `INTERNAL_SYSCALL` macro just expands to the call of the `INTERNAL_SYSCALL_NCS` macro, which prepares arguments of system call (puts them into the processor registers in correct order), executes `syscall` instruction and returns the result:

```
# define INTERNAL_SYSCALL_NCS(name, err, nr, args...)      \
  ({                                                          \
    unsigned long int resultvar;                              \
    LOAD_ARGS_##nr (args)                                      \
    LOAD_REGS_##nr                                              \
    asm volatile (                                              \
    "syscall\n\t"                                              \
    : "=a" (resultvar)                                          \
    : "0" (name) ASM_ARGS_##nr : "memory", REGISTERS_CLOBBERED_BY_SYSCALL);   \
    (long int) resultvar; })
```

The `LOAD_ARGS_##nr` macro calls the `LOAD_ARGS_N` macro where the `N` is number of arguments of the system call. In our case, it will be the `LOAD_ARGS_2` macro. Ultimately all of these macros will be expanded to the following:

```
# define LOAD_REGS_TYPES_1(t1, a1)                      \
  register t1 _a1 asm ("rdi") = __arg1;                   \
  LOAD_REGS_0

# define LOAD_REGS_TYPES_2(t1, a1, t2, a2)              \
  register t2 _a2 asm ("rsi") = __arg2;                   \
  LOAD_REGS_TYPES_1(t1, a1)
...
...
...
```

After the `syscall` instruction will be executed, the context switch will occur and the kernel will transfer execution to the system call handler. The system call handler for the `nanosleep` system call is located in the kernel/time/hrtimer.c source code file and defined with the `SYSCALL_DEFINE2` macro helper:

```
SYSCALL_DEFINE2(nanosleep, struct timespec __user *, rqtp,
        struct timespec __user *, rmtp)
{
    struct timespec tu;

    if (copy_from_user(&tu, rqtp, sizeof(tu)))
        return -EFAULT;

    if (!timespec_valid(&tu))
        return -EINVAL;

    return hrtimer_nanosleep(&tu, rmtp, HRTIMER_MODE_REL, CLOCK_MONOTONIC);
}
```

More about the `SYSCALL_DEFINE2` macro you may read in the chapter about system calls. If we look at the implementation of the `nanosleep` system call, first of all we will see that it starts from the call of the `copy_from_user` function. This function copies the given data from the userspace to kernelspace. In our case we copy timeout value to sleep to the kernelspace `timespec` structure and check that the given `timespec` is valid by the call of the `timesc_valid` function:

```
static inline bool timespec_valid(const struct timespec *ts)
{
    if (ts->tv_sec < 0)
        return false;
    if ((unsigned long)ts->tv_nsec >= NSEC_PER_SEC)
        return false;
    return true;
}
```

which just checks that the given `timespec` does not represent date before `1970` and nanoseconds does not overflow `1` second. The `nanosleep` function ends with the call of the `hrtimer_nanosleep` function from the same source code file. The `hrtimer_nanosleep` function creates a timer and calls the `do_nanosleep` function. The `do_nanosleep` does main job for us. This function provides loop:

```
do {
    set_current_state(TASK_INTERRUPTIBLE);
    hrtimer_start_expires(&t->timer, mode);

    if (likely(t->task))
        freezable_schedule();

} while (t->task && !signal_pending(current));

__set_current_state(TASK_RUNNING);
return t->task == NULL;
```

Which freezes current task during sleep. After we set `TASK_INTERRUPTIBLE` flag for the current task, the `hrtimer_start_expires` function starts the give high-resolution timer on the current processor. As the given high resolution timer will expire, the task will be again running.

That's all.

Conclusion

This is the end of the seventh part of the chapter that describes timers and timer management related stuff in the Linux kernel. In the previous part we saw x86_64 specific clock sources. As I wrote in the beginning, this part is the last part of this chapter. We saw important time management related concepts like `clocksource` and `clockevents` frameworks, `jiffies` counter and etc., in this chpater. Of course this does not cover all of the time management in the Linux kernel. Many parts of this mostly related to the scheduling which we will see in other chapter.

If you have questions or suggestions, feel free to ping me in twitter 0xAX, drop me email or just create issue.

Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-insides.

Links

- system call
- C programming language
- standard library
- glibc
- real time clock

- NTP
- nanoseconds
- register
- System V Application Binary Interface
- context switch
- Introduction to timers in the Linux kernel
- uptime
- system calls table for x86_64
- High Precision Event Timer
- Time Stamp Counter
- x86_64
- previous part

Linux 内核中的同步原语

这个章节描述内核中所有的同步原语。

- 自旋锁简介 - 这个章节的第一部分描述 Linux 内核中自旋锁机制的实现；
- 队列自旋锁 - 第二部分描述自旋锁的另一种类型 - 队列自旋锁；
- 信号量 - this part describes impmentation of `semaphore` synchronization primitive in the Linux kernel. 这个部分描述 Linux 内核中的同步原语 `semaphore` 的实现；
- 互斥锁 - 这个部分描述 Linux 内核中的 `mutex` ；
- 读者/写者信号量 - 这个部分描述特殊类型的信号量 - `reader/writer` 信号量；
- 顺序锁 - 这个部分描述 Linux 内核中的顺序锁.

Linux 内核中的同步原语. 第一部分.

Introduction

这一部分为 linux-insides 这本书开启了新的章节。定时器和时间管理相关的概念在上一个章节已经描述过了。现在是时候继续了。就像你可能从这一部分的标题所了解的那样，本章节将会描述 Linux 内核中的同步原语。

像往常一样，在考虑一些同步相关的事情之前，我们会尝试去概括地了解什么是 `同步原语` 。事实上，同步原语是一种软件机制，提供了两个或者多个并行进程或者线程在不同时刻执行一段相同的代码段的能力。例如下面的代码片段：

```
mutex_lock(&clocksource_mutex);
...
...
...
clocksource_enqueue(cs);
clocksource_enqueue_watchdog(cs);
clocksource_select();
...
...
...
mutex_unlock(&clocksource_mutex);
```

出自 kernel/time/clocksource.c 源文件。这段代码来自于 `__clocksource_register_scale` 函数，此函数添加给定的 clocksource 到时钟源列表中。这个函数在注册时钟源列表中生成两个不同的操作。例如 `clocksource_enqueue` 函数就是添加给定时钟源到注册时钟源列表——`clocksource_list` 中。注意这几行代码被两个函数所包围： `mutex_lock` 和 `mutex_unlock` ，这两个函数都带有一个参数——在本例中为 `clocksource_mutex` 。

这些函数展示了基于互斥锁 (mutex) 同步原语的加锁和解锁。当 `mutex_lock` 被执行，允许我们阻止两个或两个以上线程执行这段代码，而 `mute_unlock` 还没有被互斥锁的处理拥有者锁执行。换句话说，就是阻止在 `clocksource_list` 上的并行操作。为什么在这里需要使用 `互斥锁` ？如果两个并行处理尝试去注册一个时钟源会怎样。正如我们已经知道的那样，其中具有最大的等级（其具有最高的频率在系统中注册的时钟源）的列表中选择一个时钟源后， `clocksource_enqueue` 函数立即将一个给定的时钟源到 `clocksource_list` 列表：

```
static void clocksource_enqueue(struct clocksource *cs)
{
    struct list_head *entry = &clocksource_list;
    struct clocksource *tmp;

    list_for_each_entry(tmp, &clocksource_list, list)
        if (tmp->rating >= cs->rating)
            entry = &tmp->list;
    list_add(&cs->list, entry);
}
```

如果两个并行处理尝试同时去执行这个函数，那么这两个处理可能会找到相同的 `入口 (entry)` 可能发生竞态条件 (race condition) 或者换句话说，第二个执行 `list_add` 的处理程序，将会重写第一个线程写入的时钟源。

除了这个简答的例子，同步原语在 Linux 内核无处不在。如果再翻阅之前的[章节](https://xinqiu.gitbooks.io/linux-insides-cn/content/Timers/index.html) 或者其他章节或者如果大概看看 Linux 内核源码，就会发现许多地方都使用同步原语。我们不考虑 `mutex` 在 Linux 内核是如何实现的。事实上，Linux 内核提供了一系列不同的同步原语：

* `mutex`；
* `semaphores`；
* `seqlocks`；
* `atomic operations`；
* 等等。

现在从 `自旋锁 (spinlock)` 这个章节开始。

Linux 内核中的自旋锁。

自旋锁简单来说是一种低级的同步机制，表示了一个变量可能的两个状态：

* `acquired`；
* `released`.

每一个想要获取 `自旋锁` 的处理，必须为这个变量写入一个表示 `自旋锁获取 (spinlock acquire)` 状态的值，并且为这个变量写入 `锁释放 (spinlock released)` 状态。如果一个处理程序尝试执行受 `自旋锁` 保护的代码，那么代码将会被锁住，直到占有锁的处理程序释放掉。在本例中，所有相关的操作必须是 原子的 (atomic)，来阻止竞态条件状态。 `自旋锁` 在 Linux 内核中使用 `spinlock_t` 类型来表示。如果我们查看 Linux 内核代码，我们会看到，这个类型被 广泛地 (widely) 使用。 `spinlock_t` 的定义如下：

```
typedef struct spinlock {
        union {
              struct raw_spinlock rlock;

#ifdef CONFIG_DEBUG_LOCK_ALLOC
# define LOCK_PADSIZE (offsetof(struct raw_spinlock, dep_map))
                struct {
                        u8 __padding[LOCK_PADSIZE];
                        struct lockdep_map dep_map;
                };
#endif
        };
} spinlock_t;
```

这段代码在 include/linux/spinlock_types.h 头文件中定义。可以看出，它的实现依赖于 `CONFIG_DEBUG_LOCK_ALLOC` 内核配置选项这个状态。现在我们跳过这一块，因为所有的调试相关的事情都将会在这一部分的最后。所以，如果 `CONFIG_DEBUG_LOCK_ALLOC` 内核配置选项不可用，那么 `spinlock_t` 则包含联合体 (union)，这个联合体有一个字段—— `raw_spinlock` ：

```
typedef struct spinlock {
        union {
              struct raw_spinlock rlock;
        };
} spinlock_t;
```

`raw_spinlock` 结构的定义在相同的头文件中并且表达了 `普通 (normal)` 自旋锁的实现。让我们看看 `raw_spinlock` 结构是如何定义的：

```
typedef struct raw_spinlock {
        arch_spinlock_t raw_lock;
#ifdef CONFIG_GENERIC_LOCKBREAK
        unsigned int break_lock;
#endif
} raw_spinlock_t;
```

这里的 `arch_spinlock_t` 表示了体系结构指定的 `自旋锁` 实现并且 `break_lock` 字段持有值—— 为 `1` ，当一个处理器开始等待而锁被另一个处理器持有时，使用的对称多处理器 (SMP) 系统的例子中。这样就可以防止长时间加锁。考虑本书的 x86_64 架构，因此 `arch_spinlock_t` 被定义在 arch/x86/include/asm/spinlock_types.h 头文件中，并且看上去是这样：

```
#ifdef CONFIG_QUEUED_SPINLOCKS
#include <asm-generic/qspinlock_types.h>
#else
typedef struct arch_spinlock {
        union {
                __ticketpair_t head_tail;
                struct __raw_tickets {
                        __ticket_t head, tail;
                } tickets;
        };
} arch_spinlock_t;
```

正如我们所看到的，`arch_spinlock` 结构的定义依赖于 `CONFIG_QUEUED_SPINLOCKS` 内核配置选项的值。这个 Linux内核配置选项支持使用队列的 `自旋锁` 。这个 `自旋锁` 的特殊类型替代了 `acquired` 和 `released` 原子值，在 `队列` 上使用 `原子` 操作。如果 `CONFIG_QUEUED_SPINLOCKS` 内核配置选项启动，那么 `arch_spinlock_t` 将会被表示成如下的结构：

```
typedef struct qspinlock {
    atomic_t    val;
} arch_spinlock_t;
```

来自于 include/asm-generic/qspinlock_types.h 头文件。

目前我们不会在这个结构上停止探索，在考虑 `arch_spinlock` 和 `qspinlock` 之前，先看看自旋锁上的操作。 Linux内核在 `自旋锁` 上提供了一下主要的操作：

- `spin_lock_init` ——给定的 `自旋锁` 进行初始化；
- `spin_lock` ——获取给定的 `自旋锁` ；
- `spin_lock_bh` ——禁止软件中断并且获取给定的 `自旋锁` 。
- `spin_lock_irqsave` 和 `spin_lock_irq` ——禁止本地处理器上的中断，并且保存／不保存之前的中断状态的 `标识 (flag)` ；
- `spin_unlock` ——释放给定的 `自旋锁`；
- `spin_unlock_bh` ——释放给定的 `自旋锁` 并且启动软件中断；
- `spin_is_locked` - 返回给定的 `自旋锁` 的状态；
- 等等。

来看看 `spin_lock_init` 宏的实现。就如我已经写过的一样，这个宏和其他宏定义都在 include/linux/spinlock.h 头文件里，并且 `spin_lock_init` 宏如下所示：

```
#define spin_lock_init(_lock)         \
do {                                            \
    spinlock_check(_lock);                        \
    raw_spin_lock_init(&(_lock)->rlock);        \
} while (0)
```

正如所看到的，`spin_lock_init` 宏有一个`自旋锁`，执行两步操作：检查我们看到的给定的`自旋锁`和执行 `raw_spin_lock_init`。`spinlock_check` 的实现相当简单，实现的函数仅仅返回已知的`自旋锁`的 `raw_spinlock_t`，来确保我们精确获得`正常 (normal)`原生自旋锁：

```
static __always_inline raw_spinlock_t *spinlock_check(spinlock_t *lock)
{
    return &lock->rlock;
}
```

`raw_spin_lock_init` 宏:

```
# define raw_spin_lock_init(lock)        \
do {                                                      \
    *(lock) = __RAW_SPIN_LOCK_UNLOCKED(lock);          \
} while (0)                                               \
```

用 `__RAW_SPIN_LOCK_UNLOCKED` 的值和给定的`自旋锁`赋值给给定的 `raw_spinlock_t`。就像我们能从 `__RAW_SPIN_LOCK_UNLOCKED` 宏的名字中了解的那样，这个宏为给定的`自旋锁`执行初始化操作，并且将锁设置为`释放 (released)`状态。宏的定义在 include/linux/spinlock_types.h 头文件中，并且扩展了一下的宏：

```
#define __RAW_SPIN_LOCK_UNLOCKED(lockname)      \
         (raw_spinlock_t) __RAW_SPIN_LOCK_INITIALIZER(lockname)

#define __RAW_SPIN_LOCK_INITIALIZER(lockname)   \
         {                                                   \
             .raw_lock = __ARCH_SPIN_LOCK_UNLOCKED,          \
             SPIN_DEBUG_INIT(lockname)                       \
             SPIN_DEP_MAP_INIT(lockname)                     \
         }
```

正如之前所写的一样，我们不考虑同步原语调试相关的东西。在本例中也不考虑 `SPIN_DEBUG_INIT` 和 `SPIN_DEP_MAP_INIT` 宏。于是 `__RAW_SPINLOCK_UNLOCKED` 宏被扩展成：

```
*(&(_lock)->rlock) = __ARCH_SPIN_LOCK_UNLOCKED;
```

而 `__ARCH_SPIN_LOCK_UNLOCKED` 宏是:

```
#define __ARCH_SPIN_LOCK_UNLOCKED       { { 0 } }
```

还有:

```
#define __ARCH_SPIN_LOCK_UNLOCKED       { ATOMIC_INIT(0) }
```

这是对于 [x86_64] 架构，如果 `CONFIG_QUEUED_SPINLOCKS` 内核配置选项启用的情况。那么，在 `spin_lock_init` 宏的扩展之后，给定的 `自旋锁` 将会初始化并且状态变为—— `解锁 (unlocked)` 。

从这一时刻起我们了解了如何去初始化一个 `自旋锁` ，现在考虑 Linux 内核为 `自旋锁` 的操作提供的 API。首先是：

```
static __always_inline void spin_lock(spinlock_t *lock)
{
    raw_spin_lock(&lock->rlock);
}
```

此函数允许我们 `获取` 一个自旋锁。 `raw_spin_lock` 宏定义在同一个头文件中，并且扩展了 `_raw_spin_lock` 函数的调用：

```
#define raw_spin_lock(lock)    _raw_spin_lock(lock)
```

就像在 include/linux/spinlock.h 头文件所了解的那样， `_raw_spin_lock` 宏的定义依赖于 `CONFIG_SMP` 内核配置参数：

```
#if defined(CONFIG_SMP) || defined(CONFIG_DEBUG_SPINLOCK)
# include <linux/spinlock_api_smp.h>
#else
# include <linux/spinlock_api_up.h>
#endif
```

因此，如果在 Linux内核中 SMP 启用了，那么 `_raw_spin_lock` 宏就在 arch/x86/include/asm/spinlock.h 头文件中定义，并且看起来像这样：

```
#define _raw_spin_lock(lock) __raw_spin_lock(lock)
```

`__raw_spin_lock` 函数的定义:

```
static inline void __raw_spin_lock(raw_spinlock_t *lock)
{
        preempt_disable();
        spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);
        LOCK_CONTENDED(lock, do_raw_spin_trylock, do_raw_spin_lock);
}
```

就像你们可能了解的那样， 首先我们禁用了抢占，通过 include/linux/preempt.h (在 Linux 内核初始化进程章节的第九部分会了解到更多关于抢占)中的 `preempt_disable` 调用实现禁用。当我们将要解开给定的`自旋锁`，抢占将会再次启用：

```
static inline void __raw_spin_unlock(raw_spinlock_t *lock)
{
        ...
        ...
        ...
        preempt_enable();
}
```

当程序正在自旋锁时，这个已经获取锁的程序必须阻止其他程序方法的抢占。`spin_acquire` 宏通过其他宏宏展调用实现：

```
#define spin_acquire(l, s, t, i)                lock_acquire_exclusive(l, s, t, NULL, i)
#define lock_acquire_exclusive(l, s, t, n, i)           lock_acquire(l, s, t, 0, 1, n, i)
```

`lock_acquire` 函数:

```
void lock_acquire(struct lockdep_map *lock, unsigned int subclass,
                  int trylock, int read, int check,
                  struct lockdep_map *nest_lock, unsigned long ip)
{
         unsigned long flags;

         if (unlikely(current->lockdep_recursion))
                return;

         raw_local_irq_save(flags);
         check_flags(flags);

         current->lockdep_recursion = 1;
         trace_lock_acquire(lock, subclass, trylock, read, check, nest_lock, ip);
         __lock_acquire(lock, subclass, trylock, read, check,
                        irqs_disabled_flags(flags), nest_lock, ip, 0, 0);
         current->lockdep_recursion = 0;
         raw_local_irq_restore(flags);
}
```

就像之前所写的，我们不考虑这些调试或跟踪相关的东西。`lock_acquire` 函数的主要是通过 `raw_local_irq_save` 宏调用禁用硬件中断，因为给定的自旋锁可能被启用的硬件中断所获取。以这样的方式获取的话程序将不会被抢占。注意 `lock_acquire` 函数的最后将使用

`raw_local_irq_restore` 宏的帮助再次启动硬件中断。正如你们可能猜到的那样，主要工作将在 `__lock_acquire` 函数中定义，这个函数在 kernel/locking/lockdep.c 源代码文件中。

`__lock_acquire` 函数看起来很大。我们将试图去理解这个函数要做什么，但不是在这一部分。事实上这个函数于 Linux内核锁验证器 (lock validator) 密切相关，而这也不是此部分的主题。如果我们要返回 `__raw_spin_lock` 函数的定义，我们将会发现最终这个定义包含了以下的定义：

```
LOCK_CONTENDED(lock, do_raw_spin_trylock, do_raw_spin_lock);
```

`LOCK_CONTENDED` 宏的定义在 include/linux/lockdep.h 头文件中，而且只是使用给定 `自旋锁` 调用已知函数:

```
#define LOCK_CONTENDED(_lock, try, lock) \
         lock(_lock)
```

在本例中， `lock` 就是 include/linux/spinlock.h 头文件中的 `do_raw_spin_lock` ，而 `_lock` 就是给定的 `raw_spinlock_t` ：

```
static inline void do_raw_spin_lock(raw_spinlock_t *lock) __acquires(lock)
{
        __acquire(lock);
         arch_spin_lock(&lock->raw_lock);
}
```

这里的 `__acquire` 只是[稀疏(sparse)]相关宏，并且当前我们也对这些不感兴趣。 `arch_spin_lock` 函数定义的位置依赖于两件事：第一是系统架构，第二是我们是否使用了 `队列自旋锁(queued spinlocks)` 。本例中我们仅以 `x86_64` 架构为例介绍，因此 `arch_spin_lock` 的定义的宏表示源自 include/asm-generic/qspinlock.h 头文件中：

```
#define arch_spin_lock(l)               queued_spin_lock(l)
```

如果使用 `队列自旋锁` ，或者其他例子中， `arch_spin_lock` 函数定在 arch/x86/include/asm/spinlock.h 头文件中，如何处理？现在我们只考虑 `普通的自旋锁` ， `队列自旋锁` 相关的信息将在以后了解。来再看看 `arch_spinlock` 结构的定义，理解以下 `arch_spin_lock` 函数的实现：

```
typedef struct arch_spinlock {
         union {
                __ticketpair_t head_tail;
                struct __raw_tickets {
                        __ticket_t head, tail;
                } tickets;
        };
} arch_spinlock_t;
```

这个 `自旋锁` 的变体被称为—— `标签自旋锁 (ticket spinlock)` 。 就像我们锁了解的，标签自旋锁包括两个部分。当锁被获取，如果有程序想要获取 `自旋锁` 它就会将 `尾部(tail)` 值加1。如果 `尾部` 不等于 `头部` ， 那么程序就会被锁住，直到这些变量的值不再相等。来看看 `arch_spin_lock` 函数上的实现：

```
static __always_inline void arch_spin_lock(arch_spinlock_t *lock)
{
        register struct __raw_tickets inc = { .tail = TICKET_LOCK_INC };

        inc = xadd(&lock->tickets, inc);

        if (likely(inc.head == inc.tail))
                goto out;

        for (;;) {
                 unsigned count = SPIN_THRESHOLD;

                 do {
                       inc.head = READ_ONCE(lock->tickets.head);
                       if (__tickets_equal(inc.head, inc.tail))
                                goto clear_slowpath;
                        cpu_relax();
                 } while (--count);
                 __ticket_lock_spinning(lock, inc.tail);
         }
clear_slowpath:
        __ticket_check_and_clear_slowpath(lock, inc.head);
out:
        barrier();
}
```

`arch_spin_lock` 函数在一开始能够使用 `尾部` —— `1` 对 `__raw_tickets` 结构初始化：

```
#define __TICKET_LOCK_INC       1
```

在 `inc` 和 `lock->tickets` 的下一行执行 xadd 操作。这个操作之后 `inc` 将存储给定 `标签 (tickets)` 的值，然后 `tickets.tail` 将增加 `inc` 或 `1` 。 `尾部` 值增加 `1` 意味着一个程序开始尝试持有锁。下一步做检查，检查 `头部` 和 `尾部` 是否有相同的值。如果值相等，这意味着

没有程序持有锁并且我们去到了 `out` 标签。在 `arch_spin_lock` 函数的最后，我们可能了解了 `barrier` 宏表示 `屏障指令 (barrier instruction)`，该指令保证了编译器将不更改进入内存操作的顺序(更多关于内存屏障的知识可以阅读内核文档 (documentation))。

如果前一个程序持有锁而第二个程序开始执行 `arch_spin_lock` 函数，那么 `头部` 将不会 `等于 ``尾部`，因为 `尾部` 比 `头部` 大 `1` 。这样，程序将循环发生。在每次循环迭代的时候 `头部` 和 `尾部` 的值进行比较。如果值不相等， `cpu_relax` ，也就是 NOP 指令将会被调用：

```
#define cpu_relax()     asm volatile("rep; nop")
```

然后将开始循环的下一次迭代。如果值相等，这意味着持有锁的程序，释放这个锁并且下一个程序获取这个锁。

`spin_unlock` 操作遍布所有有 `spin_lock` 的宏或函数中，当然，使用的是 `unlock` 前缀。最后， `arch_spin_unlock` 函数将会被调用。如果看看 `arch_spin_lock` 函数的实现，我们将了解到这个函数增加了 `lock tickets` 列表的 `头部` ：

```
__add(&lock->tickets.head, TICKET_LOCK_INC, UNLOCK_LOCK_PREFIX);
```

在 `spin_lock` 和 `spin_unlock` 的组合使用中，我们得到一个队列，其 `头部` 包含了一个索引号，映射了当前执行的持有锁的程序，而 `尾部` 包含了一个索引号，映射了最后尝试持有锁的程序：

```
     +-------+       +-------+
     |       |       |       |
head |   7   | - - - |   7   | tail
     |       |       |       |
     +-------+       +-------+
                         |
                     +-------+
                     |       |
                     |   8   |
                     |       |
                     +-------+
                         |
                     +-------+
                     |       |
                     |   9   |
                     |       |
                     +-------+
```

目前这就是全部。这一部分不涵盖所有的 `自旋锁` API，但我认为这个概念背后的主要思想现在一定清楚了。

结论

涵盖 Linux 内核中的同步原语的第一部分到此结束。在这一部分，我们遇见了第一个 Linux 内核提供的同步原语 `自旋锁` 。下一部分将会继续深入这个有趣的主题，而且将会了解到其他 `同步` 相关的知识。

如果您有疑问或者建议，请在twitter 0xAX 上联系我，通过 email 联系我，或者创建一个 issue 。

友情提示：英语不是我的母语，对于译文给您带来了的不便我感到非常抱歉。如果您发现任何错误请给我发送**PR**到 **linux-insides** 。

链接

- Concurrent computing
- Synchronization
- Clocksource framework
- Mutex
- Race condition
- Atomic operations
- SMP
- x86_64
- Interrupts
- Preemption
- Linux kernel lock validator
- Sparse
- xadd instruction
- NOP
- Memory barriers
- Previous chapter

Linux 内核的同步原语. 第二部分.

队列自旋锁

这是本章节的第二部分，这部分描述 Linux 内核的和我们在本章的第一部分所见到的——自旋锁的同步原语。在这个部分我们将继续学习自旋锁的同步原语。 如果阅读了上一部分的相关内容，你可能记得除了正常自旋锁，Linux 内核还提供 `自旋锁` 的一种特殊类型 - `队列自旋锁` 。 在这个部分我们将尝试理解此概念锁代表的含义。

我们在上一部分已知 `自旋锁` 的 API:

- `spin_lock_init` - 为给定 `自旋锁` 进行初始化；
- `spin_lock` - 获取给定 `自旋锁` ；
- `spin_lock_bh` - 禁止软件中断并且获取给定 `自旋锁` ；
- `spin_lock_irqsave` 和 `spin_lock_irq` - 禁止本地处理器中断并且保存/不保存之前 `标识位` 的中断状态；
- `spin_unlock` - 释放给定的 `自旋锁` ；
- `spin_unlock_bh` - 释放给定的 `自旋锁` 并且启用软件中断；
- `spin_is_locked` - 返回给定 `自旋锁` 的状态；
- 等等。

而且我们知道所有这些宏都在 include/linux/spinlock.h 头文件中所定义，都被扩展成针对 x86_64 架构，来自于 arch/x86/include/asm/spinlock.h 文件的 `arch_spin_.*` 前缀的函数调用。如果我们关注这个头文件，我们会发现这些函数(`arch_spin_is_locked` ， `arch_spin_lock` ， `arch_spin_unlock` 等等)只在 `CONFIG_QUEUED_SPINLOCKS` 内核配置选项禁用的时才定义：

```
#ifdef CONFIG_QUEUED_SPINLOCKS
#include <asm/qspinlock.h>
#else
static __always_inline void arch_spin_lock(arch_spinlock_t *lock)
{
    ...
    ...
    ...
}
...
...
...
#endif
```

这意味着 arch/x86/include/asm/qspinlock.h 这个头文件提供提供这些函数自己的实现。实际上这些函数是宏定义并且在分布在其他头文件中。这个头文件是 include/asm-generic/qspinlock.h。如果我们查看这个头文件，我们会发现这些宏的定义：

```
#define arch_spin_is_locked(l)          queued_spin_is_locked(l)
#define arch_spin_is_contended(l)       queued_spin_is_contended(l)
#define arch_spin_value_unlocked(l)     queued_spin_value_unlocked(l)
#define arch_spin_lock(l)               queued_spin_lock(l)
#define arch_spin_trylock(l)            queued_spin_trylock(l)
#define arch_spin_unlock(l)             queued_spin_unlock(l)
#define arch_spin_lock_flags(l, f)      queued_spin_lock(l)
#define arch_spin_unlock_wait(l)        queued_spin_unlock_wait(l)
```

在我们考虑怎么排列自旋锁和实现他们的 API，我们首先看看理论部分。

介绍队列自旋锁

队列自旋锁是 Linux 内核的锁机制，是标准 `自旋锁` 的代替物。至少对 x86_64 架构是真的。如果我们查看了以下内核配置文件 - kernel/Kconfig.locks，我们将会发现以下配置入口：

```
config ARCH_USE_QUEUED_SPINLOCKS
    bool

config QUEUED_SPINLOCKS
    def_bool y if ARCH_USE_QUEUED_SPINLOCKS
    depends on SMP
```

这意味着如果 `ARCH_USE_QUEUED_SPINLOCKS` 启用，那么 `CONFIG_QUEUED_SPINLOCKS` 内核配置选项将默认启用。我们能够看到 `ARCH_USE_QUEUED_SPINLOCKS` 在 `x86_64` 特定内核配置文件 - arch/x86/Kconfig 默认开启：

```
config X86
    ...
    ...
    ...
    select ARCH_USE_QUEUED_SPINLOCKS
    ...
    ...
    ...
```

在开始考虑什么是队列自旋锁概念之前，让我们看看其他 `自旋锁` 的类型。一开始我们考虑 `正常` 自旋锁是如何实现的。通常，`正常` 自旋锁的实现是基于 test and set 指令。这个指令的工作原则真的很简单。该指令写入一个值到内存地址然后返回该地址原来的旧值。这些操作都

是在院子的上下文中完成的。也就是说，这个指令是不可中断的。因此如果第一个线程开始执行这个指令，第二个线程将会等待，直到第一个线程完成。基本锁可以在这个机制之上建立。可能看起来如下所示：

```
int lock(lock)
{
    while (test_and_set(lock) == 1)
        ;
    return 0;
}

int unlock(lock)
{
    lock=0;

    return lock;
}
```

第一个线程将执行 `test_and_set` 指令设置 `lock` 为 `1`。当第二个线程调用 `lock` 函数，它将在 `while` 循环中自旋，直到第一个线程调用 `unlock` 函数而且 `lock` 等于 `0`。这个实现对于执行不是很好，因为该实现至少有两个问题。第一个问题是该实现可能是非公平的而且一个处理器的线程可能有很长的等待时间，即使有其他线程也在等待释放锁，它还是调用了 `lock`。第二个问题是所有想要获取锁的线程，必须在共享内存的变量上执行很多类似`test_and_set` 这样的 `原子` 操作。这导致缓存失效，因为处理器缓存会存储 `lock=1`，但是在线程释放锁之后，内存中 `lock` 可能只是`1`。

在上一部分 我们了解了自旋锁的第二种实现 - `排队自旋锁(ticket spinlock)`。这一方法解决了第一个问题而且能够保证想要获取锁的线程的顺序，但是仍然存在第二个问题。

这一部分的主旨是 `队列自旋锁`。这个方法能够帮助解决上述的两个问题。`队列自旋锁`允许每个处理器对自旋过程使用他自己的内存地址。通过学习名为 MCS 锁的这种基于队列自旋锁的实现，能够最好理解基于队列自旋锁的基本原则。在了解`队列自旋锁`的实现之前，我们先尝试理解什么是 `MCS` 锁。

`MCS`锁的基本理念就在上一段已经写到了，一个线程在本地变量上自旋然后每个系统的处理器自己拥有这些变量的拷贝。换句话说这个概念建立在 Linux 内核中的 per-cpu 变量概念之上。

当第一个线程想要获取锁，线程在`队列`中注册了自身，或者换句话说，因为线程现在是闲置的，线程要加入特殊`队列`并且获取锁。当第二个线程想要在第一个线程释放锁之前获取相同锁，这个线程就会把他自身的所变量的拷贝加入到这个特殊`队列`中。这个例子中第一个线程会包含一个 `next` 字段指向第二个线程。从这一时刻，第二个线程会等待直到第一个线程释放它的锁并且关于这个事件通知给 `next` 线程。第一个线程从`队列`中删除而第二个线程持有该锁。

我们可以这样代表示意一下：

空队列：

```
+---------+
|         |
|  Queue  |
|         |
+---------+
```

第一个线程尝试获取锁：

```
+---------+     +----------------------------+
|         |     |                            |
|  Queue  |---->| First thread acquired lock |
|         |     |                            |
+---------+     +----------------------------+
```

第二个队列尝试获取锁:

```
+---------+     +----------------------------------------+     +----------------------
---+
|         |     |                                        |     |
   |
|  Queue  |---->|  Second thread waits for first thread  |<----| First thread holds lo
ck |
|         |     |                                        |     |
   |
+---------+     +----------------------------------------+     +----------------------
---+
```

或者伪代码描述为：

```
void lock(...)
{
    lock.next = NULL;
    ancestor = put_lock_to_queue_and_return_ancestor(queue, lock);

    // if we have ancestor, the lock already acquired and we
    // need to wait until it will be released
    if (ancestor)
    {
        lock.locked = 1;
        ancestor.next = lock;

        while (lock.is_locked == true)
            ;
    }

    // in other way we are owner of the lock and may exit
}

void unlock(...)
{
    // do we need to notify somebody or we are alonw in the
    // queue?
    if (lock.next != NULL) {
        // the while loop from the lock() function will be
        // finished
        lock.next.is_locked = false;
        // delete ourself from the queue and exit
        ...
        ...
        ...
        return;
    }

    // So, we have no next threads in the queue to notify about
    // lock releasing event. Let's just put `0` to the lock, will
    // delete ourself from the queue and exit.
}
```

想法很简单，但是 `队列自旋锁` 的实现一定是比伪代码复杂。就如同我上面写到的，`队列自旋锁` 机制计划在 Linux 内核中成为 `排队自旋锁` 的替代品。但你们可能还记得，常用 `自旋锁` 适用于 `32位(32-bit)` 的 字(word)。而基于 `MCS` 的锁不能使用这个大小，你们可能知道 `spinlock_t` 类型在 Linux 内核中的使用是宽字符(widely)的。这种情况下可能不得不重写 Linux 内核中重要的组成部分，但这是不可接受的。除了这一点，一些包含自旋锁用于保护的内核结构不能增长大小。但无论怎样，基于这一概念的 Linux 内核中的 `队列自旋锁` 实现有一些修改，可以适应 `32` 位的字。

这就是所有有关 `队列自旋锁` 的理论，现在让我们考虑以下在 Linux 内核中这个机制是如何实现的。 `队列自旋锁` 的实现看起来比 `排队自旋锁` 的实现更加复杂和混乱，但是细致的研究会引导成功。

队列自旋锁的API

现在我们从原理角度了解了一些 `队列自旋锁` ，是时候了解 Linux 内核中这一机制的实现了。就像我们之前了解的那样 include/asm-generic/qspinlock.h 头文件提供一套宏，代表 API 中的自旋锁的获取、释放等等。

```
#define arch_spin_is_locked(l)          queued_spin_is_locked(l)
#define arch_spin_is_contended(l)       queued_spin_is_contended(l)
#define arch_spin_value_unlocked(l)     queued_spin_value_unlocked(l)
#define arch_spin_lock(l)               queued_spin_lock(l)
#define arch_spin_trylock(l)            queued_spin_trylock(l)
#define arch_spin_unlock(l)             queued_spin_unlock(l)
#define arch_spin_lock_flags(l, f)      queued_spin_lock(l)
#define arch_spin_unlock_wait(l)        queued_spin_unlock_wait(l)
```

所有这些宏扩展了同一头文件下的函数的调用。此外，我们发现 include/asm-generic/qspinlock_types.h 头文件的 `qspinlock` 结构代表了 Linux 内核队列自旋锁。

```
typedef struct qspinlock {
    atomic_t    val;
} arch_spinlock_t;
```

如我们所了解的， `qspinlock` 结构只包含了一个字段 - `val` 。这个字段代表给定 `自旋锁` 的状态。 `4` 个字节字段包括如下 4 个部分：

- `0-7` - 上锁字节(locked byte);
- `8` - 未决位(pending bit);
- `16-17` - 这两位代表了 `MCS` 锁的 `per_cpu` 数组(马上就会了解)；
- `18-31` - 包括表明队列尾部的处理器数。

`9-15` 字节没有被使用。

就像我们已经知道的，系统中每个处理器有自己的锁拷贝。这个锁由以下结构所表示：

```
struct mcs_spinlock {
       struct mcs_spinlock *next;
       int locked;
       int count;
};
```

来自 kernel/locking/mcs_spinlock.h 头文件。第一个字段代表了指向 `队列` 中下一个线程的指针。第二个字段代表了 `队列` 中当前线程的状态，其中 `1` 是 `锁` 已经获取而 `0` 相反。然后最后一个 `mcs_spinlock` 字段 结构代表嵌套锁 (nested locks)，了解什么是嵌套锁，就像想象一下当线程已经获取锁的情况，而被硬件中断 所中断，然后中断处理程序又尝试获取锁。这个例子里，每个处理器不只是 `mcs_spinlock` 结构的拷贝，也是这些结构的数组：

```
static DEFINE_PER_CPU_ALIGNED(struct mcs_spinlock, mcs_nodes[4]);
```

此数组允许以下情况的四个事件的锁获取的四个尝试(原文：This array allows to make four attempts of a lock acquisition for the four events in following contexts:)：

- 普通任务上下文；
- 硬件中断上下文；
- 软件中断上下文；
- 屏蔽中断上下文。

现在让我们返回 `qspinlock` 结构和 `队列自旋锁` 的 `API` 中来。在我们考虑 `队列自旋锁` 的 `API` 之前，请注意 `qspinlock` 结构的 `val` 字段有类型 - `atomic_t`，此类型代表原子变量或者变量的一次操作(原文：one operation at a time variable)。一次，所有这个字段的操作都是原子的。比如说让我们看看 `val` API 的值：

```
static __always_inline int queued_spin_is_locked(struct qspinlock *lock)
{
    return atomic_read(&lock->val);
}
```

Ok，现在我们知道 Linux 内核的代表队列自旋锁数据结构，那么是时候看看 `队列自旋锁` API 中 `主要（main）` 函数的实现。

```
#define arch_spin_lock(l)               queued_spin_lock(l)
```

没错，这个函数是 - `queued_spin_lock`。正如我们可能从函数名中所了解的一样，函数允许通过线程获取锁。这个函数在 include/asm-generic/qspinlock_types.h 头文件中定义，它的实现看起来是这样：

```
static __always_inline void queued_spin_lock(struct qspinlock *lock)
{
        u32 val;

        val = atomic_cmpxchg_acquire(&lock->val, 0, _Q_LOCKED_VAL);
        if (likely(val == 0))
                return;
        queued_spin_lock_slowpath(lock, val);
}
```

看起来很简单，除了 `queued_spin_lock_slowpath` 函数，我们可能发现它只有一个参数。在我们的例子中这个参数代表 `队列自旋锁` 被上锁。让我们考虑 `队列` 锁为空，现在第一个线程想要获取锁的情况。正如我们可能了解的 `queued_spin_lock` 函数从调用 `atomic_cmpxchg_acquire` 宏开始。就像你们可能从宏的名字猜到的那样，它执行原子的 CMPXCHG 指令，使用第一个参数（当前给定自旋锁的状态）比较第二个参数（在我们的例子为零）的值，如果他们相等，那么第二个参数在存储位置保存 `_Q_LOCKED_VAL` 的值，该存储位置通过 `&lock->val` 指向并且返回这个存储位置的初始值。

`atomic_cmpxchg_acquire` 宏定义在 include/linux/atomic.h 头文件中并且扩展了 `atomic_cmpxchg` 函数的调用：

```
#define  atomic_cmpxchg_acquire         atomic_cmpxchg
```

这实现是架构所指定的。我们考虑 x86_64 架构，因此在我们的例子中这个头文件在 arch/x86/include/asm/atomic.h 并且 `atomic_cmpxchg` 函数的实现只是返回 `cmpxchg` 宏的结果：

```
static __always_inline int atomic_cmpxchg(atomic_t *v, int old, int new)
{
        return cmpxchg(&v->counter, old, new);
}
```

这个宏在arch/x86/include/asm/cmpxchg.h头文件中定义，看上去是这样：

```
#define cmpxchg(ptr, old, new) \
    __cmpxchg(ptr, old, new, sizeof(*(ptr)))

#define __cmpxchg(ptr, old, new, size) \
    __raw_cmpxchg((ptr), (old), (new), (size), LOCK_PREFIX)
```

就像我们可能了解的那样，`cmpxchg` 宏使用几乎相同的参数集合扩展了 `__cpmxchg` 宏。新添加的参数是原子值的大小。`__cpmxchg` 宏添加了 `LOCK_PREFIX`，还扩展了 `__raw_cmpxchg` 宏中 `LOCK_PREFIX` 的 LOCK指令。毕竟 `__raw_cmpxchg` 对我们来说做了所有的的工作：

```
#define __raw_cmpxchg(ptr, old, new, size, lock) \
({
    ...
    ...
    ...
    volatile u32 *__ptr = (volatile u32 *)(ptr);            \
    asm volatile(lock "cmpxchgl %2,%1"                      \
                 : "=a" (__ret), "+m" (*__ptr)              \
                 : "r" (__new), "" (__old)                  \
                 : "memory");                               \
    ...
    ...
    ...
})
```

在 `atomic_cmpxchg_acquire` 宏被执行后，该宏返回内存地址之前的值。现在只有一个线程尝试获取锁，因此 `val` 将会置为零然后我们从 `queued_spin_lock` 函数返回：

```
val = atomic_cmpxchg_acquire(&lock->val, 0, _Q_LOCKED_VAL);
if (likely(val == 0))
    return;
```

此时此刻，我们的第一个线程持有锁。注意这个行为与在 `MCS` 算法的描述有所区别。线程获取锁，但是我们不添加此线程入 `队列`。就像我之前已经写到的，`队列自旋锁` 概念的实现在 Linux 内核中基于 `MCS` 算法，但是于此同时它对优化目的有一些差异。

所以第一个线程已经获取了锁然后现在让我们考虑第二个线程尝试获取相同的锁的情况。第二个线程将从同样的 `queued_spin_lock` 函数开始，但是 `lock->val` 会包含 `1` 或者 `_Q_LOCKED_VAL`，因为第一个线程已经持有了锁。因此，在本例中 `queued_spin_lock_slowpath` 函数将会被调用。`queued_spin_lock_slowpath` 函数定义在 kernel/locking/qspinlock.c 源码文件中并且从以下的检查开始：

```
void queued_spin_lock_slowpath(struct qspinlock *lock, u32 val)
{
    if (pv_enabled())
        goto queue;

    if (virt_spin_lock(lock))
        return;

    ...
    ...
    ...
}
```

这些检查操作检查了 `pvqspinlock` 的状态。 `pvqspinlock` 是在准虚拟化（paravirtualized）环境中的 `队列自旋锁` 。就像这一章节只相关 Linux 内核同步原语一样，我们跳过这些和其他不直接相关本章节主题的部分。这些检查之后我们比较使用 `_Q_PENDING_VAL` 宏的值所代表的锁，然后什么都不做直到该比较为真（原文：After these checks we compare our value which represents lock with the value of the `_Q_PENDING_VAL` macro and do nothing while this is true）：

```
if (val == _Q_PENDING_VAL) {
    while ((val = atomic_read(&lock->val)) == _Q_PENDING_VAL)
        cpu_relax();
}
```

这里 `cpu_relax` 只是 NOP 指令。综上，我们了解了锁饱含着 - `pending` 位。这个位代表了想要获取锁的线程，但是这个锁已经被其他线程获取了，并且与此同时 `队列` 为空。在本例中， `pending` 位将被设置并且 `队列` 不会被创建(touched)。这是优化所完成的，因为不需要考虑在引发缓存无效的自身 `mcs_spinlock` 数组的创建产生的非必需隐患（原文：This is done for optimization, because there are no need in unnecessary latency which will be caused by the cache invalidation in a touching of own `mcs_spinlock` array.）。

下一步我们进入下面的循环：

```
for (;;) {
    if (val & ~_Q_LOCKED_MASK)
        goto queue;

    new = _Q_LOCKED_VAL;
    if (val == new)
        new |= _Q_PENDING_VAL;

    old = atomic_cmpxchg_acquire(&lock->val, val, new);
    if (old == val)
        break;

    val = old;
}
```

这里第一个 `if` 子句检查锁 (`val`) 的状态是上锁还是待定的(pending)。这意味着第一个线程已经获取了锁，第二个线程也试图获取锁，但现在第二个线程是待定状态。本例中我们需要开始建立队列。我们将稍后考虑这个情况。在我们的例子中，第一个线程持有锁而第二个线程也尝试获取锁。这个检查之后我们在上锁状态并且使用之前锁状态比较后创建新锁。就像你记得的那样， `val` 包含了 `&lock->val` 状态，在第二个线程调用 `atomic_cmpxchg_acquire` 宏后状态将会等于 `1` 。由于 `new` 和 `val` 的值相等，所以我们在第二个线程的锁上设置待定位。在此之后，我们需要再次检查 `&lock->val` 的值，因为第一个线程可能在这个时候释放锁。如果第一个线程还又没释放锁， `旧` 的值将等于 `val` （因为

`atomic_cmpxchg_acquire` 将会返回存储地址指向 `lock->val` 的值并且当前为 `1` ）然后我们将退出循环。因为我们退出了循环，我们会等待第一个线程直到它释放锁，清除待定位，获取锁并且返回：

```
smp_cond_acquire(!(atomic_read(&lock->val) & _Q_LOCKED_MASK));
clear_pending_set_locked(lock);
return;
```

注意我们还没创建 `队列` 。这里我们不需要，因为对于两个线程来说，队列只是导致对内存访问的非必需潜在因素。在其他的例子中，第一个线程可能在这个时候释放其锁。在本例中 `lock->val` 将包含 `_Q_LOCKED_VAL | _Q_PENDING_VAL` 并且我们会开始建立 `队列` 。通过获得处理器执行线程的本地 `mcs_nodes` 数组的拷贝我们开始建立 `队列` ：

```
node = this_cpu_ptr(&mcs_nodes[0]);
idx = node->count++;
tail = encode_tail(smp_processor_id(), idx);
```

除此之外我们计算 表示 `队列` 尾部和代表 `mcs_nodes` 数组实体的 `索引` 的 `tail` 。在此之后我们设置 `node` 指向正确的 `mcs_nodes` 数组，设置 `locked` 为零应为这个线程还没有获取锁，还有 `next` 为 `NULL` 因为我们不知道任何有关其他 `队列` 实体的信息：

```
node += idx;
node->locked = 0;
node->next = NULL;
```

我们已经创建了对于执行当前线程想获取锁的处理器的队列的 `每个 cpu（per-cpu）` 的拷贝，这意味着锁的拥有者可能在这个时刻释放了锁。因此我们可能通过 `queued_spin_trylock` 函数的调用尝试去再次获取锁。

```
if (queued_spin_trylock(lock))
        goto release;
```

`queued_spin_trylock` 函数在 include/asm-generic/qspinlock.h 头文件中被定义而且就像 `queued_spin_lock` 函数一样：

```
static __always_inline int queued_spin_trylock(struct qspinlock *lock)
{
    if (!atomic_read(&lock->val) &&
       (atomic_cmpxchg_acquire(&lock->val, 0, _Q_LOCKED_VAL) == 0))
        return 1;
    return 0;
}
```

如果锁成功被获取那么我们跳过 `释放` 标签而释放 `队列` 中的一个节点：

```
release:
    this_cpu_dec(mcs_nodes[0].count);
```

现在我们不再需要它了，因为锁已经获得了。如果 `queued_spin_trylock` 不成功，我们更新队列的尾部：

```
old = xchg_tail(lock, tail);
```

然后检索原先的尾部。下一步是检查 `队列` 是否为空。这个例子中我们需要用新的实体链接之前的实体：

```
if (old & _Q_TAIL_MASK) {
    prev = decode_tail(old);
    WRITE_ONCE(prev->next, node);

    arch_mcs_spin_lock_contended(&node->locked);
}
```

队列实体链接之后，我们开始等待直到队列的头部到来。由于我们等待头部，我们需要对可能在这个等待实践加入的新的节点做一些检查：

```
next = READ_ONCE(node->next);
if (next)
    prefetchw(next);
```

如果新节点被添加，我们从通过使用 PREFETCHW 指令指出下一个队列实体的内存中预先去除缓存线（cache line）。以优化为目的我们现在预先载入这个指针。我们只是改变了队列的头而这意味着有将要到来的 `MCS` 进行解锁操作并且下一个实体会被创建。

是的，从这个时刻我们在 `队列` 的头部。但是在我们有能力获取锁之前，我们需要至少等待两个事件：当前锁的拥有者释放锁和第二个线程处于 `待定` 位也获取锁：

```
smp_cond_acquire(!((val = atomic_read(&lock->val)) & _Q_LOCKED_PENDING_MASK));
```

两个线程都释放锁后，`队列` 的头部会持有锁。最后我们只是需要更新 `队列` 尾部然后移除从队列中移除头部。

以上。

总结

这是 Linux 内核同步原语章节第二部分的结尾。在上一个部分我们已经见到了第一个同步原语 `自旋锁` 通过 Linux 内核 实现的 `排队自旋锁（ticket spinlock）` 。在这个部分我们了解了另一个 `自旋锁` 机制的实现 - `队列自旋锁` 。下一个部分我们继续深入 Linux 内核同步原语。

如果您有疑问或者建议，请在twitter 0xAX 上联系我，通过 email 联系我，或者创建一个 issue.

友情提示：英语不是我的母语，对于译文给您带来了的不便我感到非常抱歉。如果您发现任何错误请给我发送**PR**到 **linux-insides**。

链接

- spinlock
- interrupt
- interrupt handler
- API
- Test and Set
- MCS
- per-cpu variables
- atomic instruction
- CMPXCHG instruction
- LOCK instruction
- NOP instruction
- PREFETCHW instruction
- x86_64
- Previous part

内核同步原语. 第三部分.

信号量

这是本章的第三部分 chapter，本章描述了内核中的同步原语,在之前的部分我们见到了特殊的 自旋锁 - `排队自旋锁` 。 在更前的 部分 是和 `自旋锁` 相关的描述。我们将描述更多同步原语。

在 `自旋锁` 之后的下一个我们将要讲到的 内核同步原语是 信号量。我们会从理论角度开始学习什么是 `信号量` ， 然后我们会像前几章一样讲到Linux内核是如何实现信号量的。

好吧，现在我们开始。

介绍Linux内核中的信号量

那么究竟什么是 `信号量` ？就像你可以猜到那样 - `信号量` 是另外一种支持线程或者进程的同步机制。Linux内核已经提供了一种同步机制 - `自旋锁` ， 为什么我们还需要另外一种呢？为了回答这个问题，我们需要理解这两种机制。我们已经熟悉了 `自旋锁` ,因此我们从 `信号量` 机制开始。

`自旋锁` 的设计理念是它仅会被持有非常短的时间。 但持有自旋锁的时候我们不可以进入睡眠模式因为其他的进程在等待我们。为了防止 死锁 上下文交换 也是不允许的。

当需要长时间持有一个锁的时候 信号量 就是一个很好的解决方案。从另一个方面看，这个机制对于需要短期持有锁的应用并不是最优。为了理解这个问题，我们需要知道什么是 `信号量` 。

就像一般的同步原语， `信号量` 是基于变量的。这个变量可以变大或者减少，并且这个变量的状态代表了获取锁的能力。注意这个变量的值并不限于 `0` 和 `1` 。有两种类型的 `信号量` ：

- `二值信号量`;
- `普通信号量`.

第一种 `信号量` 的值可以为 `1` 或者 `0` 。第二种 `信号量` 的值可以为任何非负数。如果 `信号量` 的值大于 `1` 那么它被叫做 `计数信号量` ，并且它允许多于 `1` 个进程获取它。这种机制允许我们记录现有的资源，而 `自旋锁` 只允许我们为一个任务上锁。除了所有这些之外，另外一个重要的点是 `信号量` 允许进入睡眠状态。 另外当某进程在等待一个被其他进程获取的锁时， 调度器 也许会切换别的进程。

信号量 API

因此，我们从理论方面了解一些 `信号量` 的知识，我们来看看它在Linux内核中是如何实现的。所有 `信号量` 相关的 API 都在名为 include/linux/semaphore.h 的头文件中

我们看到 `信号量` 机制是有以下的结构体表示的：

```
struct semaphore {
    raw_spinlock_t        lock;
    unsigned int        count;
    struct list_head    wait_list;
};
```

在内核中， `信号量` 结构体由三部分组成：

- `lock` - 保护 `信号量` 的 `自旋锁` ;
- `count` - 现有资源的数量;
- `wait_list` - 等待获取此锁的进程序列.

在我们考虑Linux内核的的 `信号量` API 之前，我们需要知道如何初始化一个 `信号量` 。事实上， Linux内核提供了两个 `信号量` 的初始函数。这些函数允许初始化一个 `信号量` 为：

- `静态` ;
- `动态` .

我们来看看第一个种初始化静态 `信号量` 。我们可以使用 `DEFINE_SEMAPHORE` 宏将 `信号量` 静态初始化。

```
#define DEFINE_SEMAPHORE(name)  \
         struct semaphore name = __SEMAPHORE_INITIALIZER(name, 1)
```

就像我们看到这样， `DEFINE_SEMAPHORE` 宏只提供了初始化 `二值` 信号量。 `DEFINE_SEMAPHORE` 宏展开到 `信号量` 结构体的定义。结构体通过 `__SEMAPHORE_INITIALIZER` 宏初始化。我们来看看这个宏的实现

```
#define __SEMAPHORE_INITIALIZER(name, n)              \
{                                                                    \
        .lock           = __RAW_SPIN_LOCK_UNLOCKED((name).lock),     \
        .count          = n,                                         \
        .wait_list      = LIST_HEAD_INIT((name).wait_list),          \
}
```

`__SEMAPHORE_INITIALIZER` 宏传入了 `信号量` 结构体的名字并且初始化这个结构体的各个域。首先我们使用 `__RAW_SPIN_LOCK_UNLOCKED` 宏对给予的 `信号量` 初始化一个 `自旋锁` 。就像你从之前 的部分看到那样， `__RAW_SPIN_LOCK_UNLOCKED` 宏是在 include/linux/spinlock_types.h 头文件中定义，它展开到 `__ARCH_SPIN_LOCK_UNLOCKED` 宏，而 `__ARCH_SPIN_LOCK_UNLOCKED` 宏又展开到零或者无锁状态

```
#define __ARCH_SPIN_LOCK_UNLOCKED       { { 0 } }
```

`信号量` 的最后两个域 `count` 和 `wait_list` 是通过现有资源的数量和空 链表来初始化。 第二种初始化 `信号量` 的方式是将 `信号量` 和现有资源数目传送给 `sema_init` 函数。 这个函数是在 include/linux/semaphore.h 头文件中定义的。

```
static inline void sema_init(struct semaphore *sem, int val)
{
        static struct lock_class_key __key;
        *sem = (struct semaphore) __SEMAPHORE_INITIALIZER(*sem, val);
        lockdep_init_map(&sem->lock.dep_map, "semaphore->lock", &__key, 0);
}
```

我们来看看这个函数是如何实现的。它看起来很简单。函数使用我们刚看到的 `__SEMAPHORE_INITIALIZER` 宏对传入的 `信号量` 进行初始化。就像我们在之前 部分 写的那样，我们将会跳过Linux内核关于 锁验证 的部分。 从现在开始我们知道如何初始化一个 `信号量`，我们看看如何上锁和解锁。Linux内核提供了如下操作 `信号量` 的 API

```
void down(struct semaphore *sem);
void up(struct semaphore *sem);
int  down_interruptible(struct semaphore *sem);
int  down_killable(struct semaphore *sem);
int  down_trylock(struct semaphore *sem);
int  down_timeout(struct semaphore *sem, long jiffies);
```

前两个函数： `down` 和 `up` 是用来获取或释放 `信号量`。 `down_interruptible` 函数试图去获取一个 `信号量`。如果被成功获取，`信号量` 的计数就会被减少并且锁也会被获取。同时当前任务也会被调度到受阻状态，也就是说 `TASK_INTERRUPTIBLE` 标志将会被至位。`TASK_INTERRUPTIBLE` 表示这个进程也许可以通过 信号 退回到销毁状态。

`down_killable` 函数和 `down_interruptible` 函数提供类似的功能，但是它还将当前进程的 `TASK_KILLABLE` 标志置位。这表示等待的进程可以被杀死信号中断。

`down_trylock` 函数和 `spin_trylock` 函数相似。这个函数试图去获取一个锁并且退出如果这个操作是失败的。在这个例子中，想获取锁的进程不会等待。最后的 `down_timeout` 函数试图去获取一个锁。当前进程将会被中断进入到等待状态当超过传入的可等待时间。除此之外你也许注意到，这个等待的时间是以 jiffies计数。

我们刚刚看了 `信号量` API的定义。我们从 `down` 函数开始看。这个函数是在 kernel/locking/semaphore.c 源代码定义的。我们来看看函数实现：

```
void down(struct semaphore *sem)
{
        unsigned long flags;

        raw_spin_lock_irqsave(&sem->lock, flags);
        if (likely(sem->count > 0))
                sem->count--;
        else
                __down(sem);
        raw_spin_unlock_irqrestore(&sem->lock, flags);
}
EXPORT_SYMBOL(down);
```

我们先看在 `down` 函数起始处定义的 `flags` 变量。这个变量将会传入到 `raw_spin_lock_irqsave` 和 `raw_spin_lock_irqrestore` 宏定义。这些宏是在 include/linux/spinlock.h头文件定义的。这些宏用来保护当前 `信号量` 的计数器。事实上这两个宏的作用和 `spin_lock` 和 `spin_unlock` 宏相似。只不过这组宏会存储/重置当前中断标志并且禁止 中断 。

就像你猜到那样， `down` 函数的主要就是通过 `raw_spin_lock_irqsave` 和 `raw_spin_unlock_irqrestore` 宏来实现的。我们通过将 `信号量` 的计数器和零对比，如果计数器大于零，我们可以减少这个计数器。这表示我们已经获取了这个锁。否则如果计数器是零，这表示所以的现有资源都已经被占用，我们需要等待以获取这个锁。就像我们看到那样， `__down` 函数将会被调用。 `__down` 函数是在 相同)的源代码定义的，它的实现看起来如下：

```
static noinline void __sched __down(struct semaphore *sem)
{
        __down_common(sem, TASK_UNINTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
}
```

`__down` 函数仅仅调用了 `__down_common` 函数，并且传入了三个参数

- `semaphore` ;
- `flag` - 对当前任务;
- `timeout` - 最长等待 `信号量` 的时间.

在我们看 `__down_common` 函数之前，注意 `down_trylock` , `down_timeout` 和 `down_killable` 的实现也都是基于 `__down_common` 函数。

```
static noinline int __sched __down_interruptible(struct semaphore *sem)
{
        return __down_common(sem, TASK_INTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
}
```

`__down_killable` 函数：

```
static noinline int __sched __down_killable(struct semaphore *sem)
{
        return __down_common(sem, TASK_KILLABLE, MAX_SCHEDULE_TIMEOUT);
}
```

`__down_timeout` 函数:

```
static noinline int __sched __down_timeout(struct semaphore *sem, long timeout)
{
        return __down_common(sem, TASK_UNINTERRUPTIBLE, timeout);
}
```

现在我们来看看 `__down_common` 函数的实现。这个函数是在 kernel/locking/semaphore.c源文件中定义的。这个函数的定义从以下两个本地变量开始。

```
struct task_struct *task = current;
struct semaphore_waiter waiter;
```

第一个变量表示当前想获取本地处理器锁的任务。 `current` 宏是在 arch/x86/include/asm/current.h 头文件中定义的。

```
#define current get_current()
```

`get_current` 函数返回 `current_task` per-cpu 变量的值。

```
DECLARE_PER_CPU(struct task_struct *, current_task);

static __always_inline struct task_struct *get_current(void)
{
        return this_cpu_read_stable(current_task);
}
```

第二个变量是 `waiter` 表示了一个 `semaphore.wait_list` 列表的入口：

```
struct semaphore_waiter {
        struct list_head list;
        struct task_struct *task;
        bool up;
};
```

下一步我们将当前进程加入到 `wait_list` 并且在定义如下变量后填充 `waiter` 域

```
list_add_tail(&waiter.list, &sem->wait_list);
waiter.task = task;
waiter.up = false;
```

下一步我们进入到如下的无限循环：

```
for (;;) {
        if (signal_pending_state(state, task))
            goto interrupted;

        if (unlikely(timeout <= 0))
            goto timed_out;

        __set_task_state(task, state);

        raw_spin_unlock_irq(&sem->lock);
        timeout = schedule_timeout(timeout);
        raw_spin_lock_irq(&sem->lock);

        if (waiter.up)
            return 0;
}
```

在之前的代码中我们将 `waiter.up` 设置为 `false`。所以当 `up` 没有设置为 `true` 任务将会在这个无限循环中循环。这个循环从检查当前的任务是否处于 `pending` 状态开始，也就是说此任务的标志包含 `TASK_INTERRUPTIBLE` 或者 `TASK_WAKEKILL` 标志。我之前写到当一个任务在等待获取一个信号的时候任务也许可以被〔信号〕(https://en.wikipedia.org/wiki/Unix_signal)中断。`signal_pending_state` 函数是在 include/linux/sched.h原文件中定义的，它看起来如下：

```
static inline int signal_pending_state(long state, struct task_struct *p)
{
         if (!(state & (TASK_INTERRUPTIBLE | TASK_WAKEKILL)))
                 return 0;
         if (!signal_pending(p))
                 return 0;

         return (state & TASK_INTERRUPTIBLE) || __fatal_signal_pending(p);
}
```

我们先会检测 `state` 位掩码 包含 `TASK_INTERRUPTIBLE` 或者 `TASK_WAKEKILL` 位，如果不包含这两个位，函数退出。下一步我们检测当前任务是否有一个挂起信号，如果没有挂起信号函数退出。最后我们就检测 `state` 位掩码的 `TASK_INTERRUPTIBLE` 位。如果，我们任务包含一个挂起信号，我们将会跳转到 `interrupted` 标签：

```
interrupted:
    list_del(&waiter.list);
    return -EINTR;
```

在这个标签中，我们会删除等待锁的列表，然后返回 `-EINTR` 错误码。 如果一个任务没有挂起信号，我们检测超时是否小于等于零。

```
if (unlikely(timeout <= 0))
    goto timed_out;
```

我们跳转到 `timed_out` 标签：

```
timed_out:
    list_del(&waiter.list);
    return -ETIME;
```

在这个标签里，我们继续做和 `interrupted` 一样的事情。我们将任务从锁等待者中删除，但是返回 `-ETIME` 错误码。如果一个任务没有挂起信号而且给予的超时也没有过期，当前的任务将会被设置为传入的 `state` ：

```
__set_task_state(task, state);
```

然后调用 `schedule_timeout` 函数：

```
raw_spin_unlock_irq(&sem->lock);
timeout = schedule_timeout(timeout);
raw_spin_lock_irq(&sem->lock);
```

这个函数是在 kernel/time/timer.c 代码中定义的。 `schedule_timeout` 函数将当前的任务置为休眠到设置的超时为止。

这就是所有关于 `__down_common` 函数。如果一个函数想要获取一个已经被其它任务获取的锁，它将会转入到无限循环。并且它不能被信号中断，当前设置的超时不会过期或者当前持有锁的任务不释放它。现在我们来看看 `up` 函数的实现。

`up` 函数和 `down` 函数定义在同一个 原文件。这个函数的主要功能是释放锁，这个函数看起来：

```
void up(struct semaphore *sem)
{
        unsigned long flags;

        raw_spin_lock_irqsave(&sem->lock, flags);
        if (likely(list_empty(&sem->wait_list)))
                sem->count++;
        else
                __up(sem);
        raw_spin_unlock_irqrestore(&sem->lock, flags);
}
EXPORT_SYMBOL(up);
```

它看起来和 `down` 函数相似。这里有两个不同点。首先我们增加 `semaphore` 的计数。如果等待列表是空的，我们调用在当前原文件中定义的 `__up` 函数。如果等待列表不是空的，我们需要允许列表中的第一个任务去获取一个锁：

```
static noinline void __sched __up(struct semaphore *sem)
{
        struct semaphore_waiter *waiter = list_first_entry(&sem->wait_list,
                                                struct semaphore_waiter, list);
        list_del(&waiter->list);
        waiter->up = true;
        wake_up_process(waiter->task);
}
```

在此我们获取待序列中的第一个任务，将它从列表中删除，将它的 `waiter-up` 设置为真。从此刻起 `__down_common` 函数中的无限循环将会被停止。 `wake_up_process` 函数将会在 `__up` 函数的结尾调用。我们从 `__down_common` 函数调用的 `schedule_timeout` 函数调用了 `schedule_timeout` 函数。 `schedule_timeout` 函数将当前任务置于睡眠状态直到超时等待。现在我们进程也许会睡眠，我们需要唤醒。这就是为什么我们需要从 kernel/sched/core.c 源代码中调用 `wake_up_process` 函数

这就是所有的信息了。

小结

这就是Linux内核中关于 同步原语 的第三部分的终结。在之前的两个部分，我们已经见到了第一个Linux内核的同步原语 `自旋锁` ，它是使用 `ticket spinlock` 实现并且用于很短时间的锁。在这个部分我们见到了另外一种同步原语 — 信号量，信号量用于长时间的锁，因为它会导致 上下文切换。 在下一部分，我们将会继续深入Linux内核的同步原语并且讨论另一个同步原语 — 互斥量。

如果你有问题或者建议，请在twitter 0xAX上联系我，通过 email联系我，或者创建一个 issue

链接

- spinlocks
- synchronization primitive
- semaphore
- context switch
- preemption
- deadlocks
- scheduler
- Doubly linked list in the Linux kernel
- jiffies
- interrupts
- per-cpu
- bitmask
- SIGKILL
- errno
- API
- mutex
- Previous part

Synchronization primitives in the Linux kernel. Part 4.

Introduction

This is the fourth part of the chapter which describes synchronization primitives in the Linux kernel and in the previous parts we finished to consider different types spinlocks and semaphore synchronization primitives. We will continue to learn synchronization primitives in this part and consider yet another one which is called - mutex which is stands for `MUTual EXclusion`.

As in all previous parts of this book, we will try to consider this synchronization primitive from the theoretical side and only than we will consider API provided by the Linux kernel to manipulate with `mutexes`.

So, let's start.

Concept of `mutex`

We already familiar with the semaphore synchronization primitive from the previous part. It represented by the:

```
struct semaphore {
    raw_spinlock_t        lock;
    unsigned int        count;
    struct list_head    wait_list;
};
```

structure which holds information about state of a lock and list of a lock waiters. Depends on the value of the `count` field, a `semaphore` can provide access to a resource of more than one wishing of this resource. The mutex concept is very similar to a semaphore concept. But it has some differences. The main difference between `semaphore` and `mutex` synchronization primitive is that `mutex` has more strict semantic. Unlike a `semaphore`, only one process may hold `mutex` at one time and only the `owner` of a `mutex` may release or unlock it. Additional difference in implementation of `lock` API. The `semaphore` synchronization primitive forces rescheduling of processes which are in waiters list. The implementation of `mutex` lock `API` allows to avoid this situation and as a result expensive context switches.

The `mutex` synchronization primitive represented by the following:

```
struct mutex {
        atomic_t                count;
        spinlock_t              wait_lock;
        struct list_head        wait_list;
#if defined(CONFIG_DEBUG_MUTEXES) || defined(CONFIG_MUTEX_SPIN_ON_OWNER)
        struct task_struct      *owner;
#endif
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
        struct optimistic_spin_queue osq;
#endif
#ifdef CONFIG_DEBUG_MUTEXES
        void                    *magic;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
        struct lockdep_map      dep_map;
#endif
};
```

structure in the Linux kernel. This structure is defined in the include/linux/mutex.h header file and contains similar to the `semaphore` structure set of fields. The first field of the `mutex` structure is - `count`. Value of this field represents state of a `mutex`. In a case when the value of the `count` field is `1`, a `mutex` is in `unlocked` state. When the value of the `count` field is `zero`, a `mutex` is in the `locked` state. Additionally value of the `count` field may be `negative`. In this case a `mutex` is in the `locked` state and has possible waiters.

The next two fields of the `mutex` structure - `wait_lock` and `wait_list` are spinlock for the protection of a `wait queue` and list of waiters which represents this `wait queue` for a certain lock. As you may notice, the similarity of the `mutex` and `semaphore` structures ends. Remaining fields of the `mutex` structure, as we may see depends on different configuration options of the Linux kernel.

The first field - `owner` represents process which acquired a lock. As we may see, existence of this field in the `mutex` structure depends on the `CONFIG_DEBUG_MUTEXES` or `CONFIG_MUTEX_SPIN_ON_OWNER` kernel configuration options. Main point of this field and the next `osq` fields is support of `optimistic spinning` which we will see later. The last two fields - `magic` and `dep_map` are used only in debugging mode. The `magic` field is to storing a `mutex` related information for debugging and the second field - `lockdep_map` is for lock validator of the Linux kernel.

Now, after we have considered the `mutex` structure, we may consider how this synchronization primitive works in the Linux kernel. As you may guess, a process which wants to acquire a lock, must to decrease value of the `mutex->count` if possible. And if a

process wants to release a lock, it must to increase the same value. That's true. But as you may also guess, it is not so simple in the Linux kernel.

Actually, when a process try to acquire a `mutex`, there three possible paths:

- `fastpath`;
- `midpath`;
- `slowpath`.

which may be taken, depending on the current state of the `mutex`. The first path or `fastpath` is the fastest as you may understand from its name. Everything is easy in this case. Nobody acquired a `mutex`, so the value of the `count` field of the `mutex` structure may be directly decremented. In a case of unlocking of a `mutex`, the algorithm is the same. A process just increments the value of the `count` field of the `mutex` structure. Of course, all of these operations must be atomic.

Yes, this looks pretty easy. But what happens if a process wants to acquire a `mutex` which is already acquired by other process? In this case, the control will be transferred to the second path - `midpath`. The `midpath` or `optimistic spinning` tries to spin with already familiar for us MCS lock while the lock owner is running. This path will be executed only if there are no other processes ready to run that have higher priority. This path is called `optimistic` because the waiting task will not be sleep and rescheduled. This allows to avoid expensive context switch.

In the last case, when the `fastpath` and `midpath` may not be executed, the last path - `slowpath` will be executed. This path acts like a semaphore lock. If the lock is unable to be acquired by a process, this process will be added to `wait queue` which is represented by the following:

```
struct mutex_waiter {
        struct list_head        list;
        struct task_struct      *task;
#ifdef CONFIG_DEBUG_MUTEXES
        void                    *magic;
#endif
};
```

structure from the include/linux/mutex.h header file and will be sleep. Before we will consider API which is provided by the Linux kernel for manipulation with `mutexes`, let's consider the `mutex_waiter` structure. If you have read the previous part of this chapter, you may notice that the `mutex_waiter` structure is similar to the `semaphore_waiter` structure from the kernel/locking/semaphore.c source code file:

```
struct semaphore_waiter {
        struct list_head list;
        struct task_struct *task;
        bool up;
};
```

It also contains `list` and `task` fields which are represent entry of the mutex wait queue. The one difference here that the `mutex_waiter` does not contains `up` field, but contains the `magic` field which depends on the `CONFIG_DEBUG_MUTEXES` kernel configuration option and used to store a `mutex` related information for debugging purpose.

Now we know what is it `mutex` and how it is represented the Linux kernel. In this case, we may go ahead and start to look at the API which the Linux kernel provides for manipulation of `mutexes`.

Mutex API

Ok, in the previous paragraph we knew what is it `mutex` synchronization primitive and saw the `mutex` structure which represents `mutex` in the Linux kernel. Now it's time to consider API for manipulation of mutexes. Description of the `mutex` API is located in the include/linux/mutex.h header file. As always, before we will consider how to acquire and release a `mutex`, we need to know how to initialize it.

There are two approaches to initialize a `mutex`. The first is to do it statically. For this purpose the Linux kernel provides following:

```
#define DEFINE_MUTEX(mutexname) \
        struct mutex mutexname = __MUTEX_INITIALIZER(mutexname)
```

macro. Let's consider implementation of this macro. As we may see, the `DEFINE_MUTEX` macro takes name for the `mutex` and expands to the definition of the new `mutex` structure. Additionally new `mutex` structure get initialized with the `__MUTEX_INITIALIZER` macro. Let's look at the implementation of the `__MUTEX_INITIALIZER`:

```
#define __MUTEX_INITIALIZER(lockname)         \
{                                                             \
       .count = ATOMIC_INIT(1),                               \
       .wait_lock = __SPIN_LOCK_UNLOCKED(lockname.wait_lock), \
       .wait_list = LIST_HEAD_INIT(lockname.wait_list)        \
}
```

This macro is defined in the same header file and as we may understand it initializes fields of the `mutex` structure the initial values. The `count` field get initialized with the `1` which represents `unlocked` state of a mutex. The `wait_lock` spinlock get initialized to the unlocked state and the last field `wait_list` to empty doubly linked list.

The second approach allows us to initialize a `mutex` dynamically. To do this we need to call the `__mutex_init` function from the kernel/locking/mutex.c source code file. Actually, the `__mutex_init` function rarely called directly. Instead of the `__mutex_init`, the:

```
# define mutex_init(mutex)                \
do {                                                    \
        static struct lock_class_key __key;             \
                                                        \
        __mutex_init((mutex), #mutex, &__key);          \
} while (0)
```

macro is used. We may see that the `mutex_init` macro just defines the `lock_class_key` and call the `__mutex_init` function. Let's look at the implementation of this function:

```
void
__mutex_init(struct mutex *lock, const char *name, struct lock_class_key *key)
{
        atomic_set(&lock->count, 1);
        spin_lock_init(&lock->wait_lock);
        INIT_LIST_HEAD(&lock->wait_list);
        mutex_clear_owner(lock);
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
        osq_lock_init(&lock->osq);
#endif
        debug_mutex_init(lock, name, key);
}
```

As we may see the `__mutex_init` function takes three arguments:

- `lock` - a mutex itself;
- `name` - name of mutex for debugging purpose;
- `key` - key for lock validator.

At the beginning of the `__mutex_init` function, we may see initialization of the `mutex` state. We set it to `unlocked` state with the `atomic_set` function which atomically set the give variable to the given value. After this we may see initialization of the `spinlock` to the unlocked state which will protect `wait queue` of the `mutex` and initialization of the `wait queue` of the `mutex`. After this we clear owner of the `lock` and initialize optimistic queue by the call of the `osq_lock_init` function from the include/linux/osq_lock.h header file. This function just sets the tail of the optimistic queue to the unlocked state:

```
static inline bool osq_is_locked(struct optimistic_spin_queue *lock)
{
        return atomic_read(&lock->tail) != OSQ_UNLOCKED_VAL;
}
```

In the end of the `__mutex_init` function we may see the call of the `debug_mutex_init` function, but as I already wrote in previous parts of this chapter, we will not consider debugging related stuff in this chapter.

After the `mutex` structure is initialized, we may go ahead and will look at the `lock` and `unlock` API of `mutex` synchronization primitive. Implementation of `mutex_lock` and `mutex_unlock` functions located in the kernel/locking/mutex.c source code file. First of all let's start from the implementation of the `mutex_lock`. It looks:

```
void __sched mutex_lock(struct mutex *lock)
{
        might_sleep();
        __mutex_fastpath_lock(&lock->count, __mutex_lock_slowpath);
        mutex_set_owner(lock);
}
```

We may see the call of the `might_sleep` macro from the include/linux/kernel.h header file at the beginning of the `mutex_lock` function. Implementation of this macro depends on the `CONFIG_DEBUG_ATOMIC_SLEEP` kernel configuration option and if this option is enabled, this macro just prints a stack trace if it was executed in atomic context. This macro is helper for debugging purposes. In other way this macro does nothing.

After the `might_sleep` macro, we may see the call of the `__mutex_fastpath_lock` function. This function is architecture-specific and as we consider x86_64 architecture in this book, the implementation of the `__mutex_fastpath_lock` is located in the arch/x86/include/asm/mutex_64.h header file. As we may understand from the name of the `__mutex_fastpath_lock` function, this function will try to acquire lock in a fast path or in other words this function will try to decrement the value of the `count` of the given mutex.

Implementation of the `__mutex_fastpath_lock` function consists from two parts. The first part is inline assembly statement. Let's look at it:

```
asm_volatile_goto(LOCK_PREFIX "   decl %0\n"
                              "   jns %l[exit]\n"
                              : : "m" (v->counter)
                              : "memory", "cc"
                              : exit);
```

First of all, let's pay attention to the `asm_volatile_goto`. This macro is defined in the include/linux/compiler-gcc.h header file and just expands to the two inline assembly statements:

```
#define asm_volatile_goto(x...) do { asm goto(x); asm (""); } while (0)
```

The first assembly statement contains `goto` specificator and the second empty inline assembly statement is barrier. Now let's return to the our inline assembly statement. As we may see it starts from the definition of the `LOCK_PREFIX` macro which just expands to the lock instruction:

```
#define LOCK_PREFIX LOCK_PREFIX_HERE "\n\tlock; "
```

As we already know from the previous parts, this instruction allows to execute prefixed instruction atomically. So, at the first step in the our assembly statement we try decrement value of the given `mutex->counter`. At the next step the jns instruction will execute jump at the `exit` label if the value of the decremented `mutex->counter` is not negative. The `exit` label is the second part of the `__mutex_fastpath_lock` function and it just points to the exit from this function:

```
exit:
        return;
```

For this moment he implementation of the `__mutex_fastpath_lock` function looks pretty easy. But the value of the `mutex->counter` may be negative after increment. In this case the:

```
fail_fn(v);
```

will be called after our inline assembly statement. The `fail_fn` is the second parameter of the `__mutex_fastpath_lock` function and represents pointer to function which represents `midpath/slowpath` paths to acquire the given lock. In our case the `fail_fn` is the `__mutex_lock_slowpath` function. Before we will look at the implementation of the `__mutex_lock_slowpath` function, let's finish with the implementation of the `mutex_lock` function. In the simplest way, the lock will be acquired successfully by a process and the `__mutex_fastpath_lock` will be finished. In this case, we just call the

```
mutex_set_owner(lock);
```

in the end of the `mutex_lock`. The `mutex_set_owner` function is defined in the kernel/locking/mutex.h header file and just sets owner of a lock to the current process:

```
static inline void mutex_set_owner(struct mutex *lock)
{
        lock->owner = current;
}
```

In other way, let's consider situation when a process which wants to acquire a lock is unable to do it, because another process already acquired the same lock. We already know that the `__mutex_lock_slowpath` function will be called in this case. Let's consider implementation of this function. This function is defined in the kernel/locking/mutex.c source code file and starts from the obtaining of the proper mutex by the mutex state given from the `__mutex_fastpath_lock` with the `container_of` macro:

```
__visible void __sched
__mutex_lock_slowpath(atomic_t *lock_count)
{
        struct mutex *lock = container_of(lock_count, struct mutex, count);

        __mutex_lock_common(lock, TASK_UNINTERRUPTIBLE, 0,
                            NULL, _RET_IP_, NULL, 0);
}
```

and call the `__mutex_lock_common` function with the obtained `mutex`. The `__mutex_lock_common` function starts from preemtion disabling until rescheduling:

```
preempt_disable();
```

After this comes the stage of optimistic spinning. As we already know this stage depends on the `CONFIG_MUTEX_SPIN_ON_OWNER` kernel configuration option. If this option is disabled, we skip this stage and move at the last path - `slowpath` of a `mutex` acquisition:

```
if (mutex_optimistic_spin(lock, ww_ctx, use_ww_ctx)) {
        preempt_enable();
        return 0;
}
```

First of all the `mutex_optimistic_spin` function check that we don't need to reschedule or in other words there are no other tasks ready to run that have higher priority. If this check was successful we need to update `MCS` lock wait queue with the current spin. In this way only one spinner can complete for the mutex at one time:

```
osq_lock(&lock->osq)
```

At the next step we start to spin in the next loop:

```
while (true) {
    owner = READ_ONCE(lock->owner);

    if (owner && !mutex_spin_on_owner(lock, owner))
        break;

    if (mutex_try_to_acquire(lock)) {
        lock_acquired(&lock->dep_map, ip);

        mutex_set_owner(lock);
        osq_unlock(&lock->osq);
        return true;
    }
}
```

and try to acquire a lock. First of all we try to take current owner and if the owner exists (it may not exists in a case when a process already released a mutex) and we wait for it in the `mutex_spin_on_owner` function before the owner will release a lock. If new task with higher priority have appeared during wait of the lock owner, we break the loop and go to sleep. In other case, the process already may release a lock, so we try to acquire a lock with the `mutex_try_to_acquired` . If this operation finished successfully, we set new owner for the given mutex, removes ourself from the `MCS` wait queue and exit from the `mutex_optimistic_spin` function. At this state a lock will be acquired by a process and we enable preemtion and exit from the `__mutex_lock_common` function:

```
if (mutex_optimistic_spin(lock, ww_ctx, use_ww_ctx)) {
    preempt_enable();
    return 0;
}
```

That's all for this case.

In other case all may not be so successful. For example new task may occur during we spinning in the loop from the `mutex_optimistic_spin` or even we may not get to this loop from the `mutex_optimistic_spin` in a case when there were task(s) with higher priority before this loop. Or finally the `CONFIG_MUTEX_SPIN_ON_OWNER` kernel configuration option disabled. In this case the `mutex_optimistic_spin` will do nothing:

```
#ifndef CONFIG_MUTEX_SPIN_ON_OWNER
static bool mutex_optimistic_spin(struct mutex *lock,
                                  struct ww_acquire_ctx *ww_ctx, const bool use_ww_ctx)

{
    return false;
}
#endif
```

In all of these cases, the `__mutex_lock_common` function will acct like a `semaphore`. We try to acquire a lock again because the owner of a lock might already release a lock before this time:

```
if (!mutex_is_locked(lock) &&
   (atomic_xchg_acquire(&lock->count, 0) == 1))
      goto skip_wait;
```

In a failure case the process which wants to acquire a lock will be added to the waiters list

```
list_add_tail(&waiter.list, &lock->wait_list);
waiter.task = task;
```

In a successful case we update the owner of a lock, enable preemption and exit from the `__mutex_lock_common` function:

```
skip_wait:
        mutex_set_owner(lock);
        preempt_enable();
        return 0;
```

In this case a lock will be acquired. If can't acquire a lock for now, we enter into the following loop:

```
for (;;) {

    if (atomic_read(&lock->count) >= 0 && (atomic_xchg_acquire(&lock->count, -1) == 1)
)
        break;

    if (unlikely(signal_pending_state(state, task))) {
        ret = -EINTR;
        goto err;
    }

    __set_task_state(task, state);

     schedule_preempt_disabled();
}
```

where try to acquire a lock again and exit if this operation was successful. Yes, we try to acquire a lock again right after unsuccessful try before the loop. We need to do it to make sure that we get a wakeup once a lock will be unlocked. Besides this, it allows us to acquire a lock after sleep. In other case we check the current process for pending signals and exit if the process was interrupted by a `signal` during wait for a lock acquisition. In the end of loop we didn't acquire a lock, so we set the task state for `TASK_UNINTERRUPTIBLE` and go to sleep with call of the `schedule_preempt_disabled` function.

That's all. We have considered all three possible paths through which a process may pass when it will wan to acquire a lock. Now let's consider how `mutex_unlock` is implemented. When the `mutex_unlock` will be called by a process which wants to release a lock, the `__mutex_fastpath_unlock` will be called from the arch/x86/include/asm/mutex_64.h header file:

```
void __sched mutex_unlock(struct mutex *lock)
{
    __mutex_fastpath_unlock(&lock->count, __mutex_unlock_slowpath);
}
```

Implementation of the `__mutex_fastpath_unlock` function is very similar to the implementation of the `__mutex_fastpath_lock` function:

```
static inline void __mutex_fastpath_unlock(atomic_t *v,
                                           void (*fail_fn)(atomic_t *))
{
        asm_volatile_goto(LOCK_PREFIX "   incl %0\n"
                          "   jg %l[exit]\n"
                          : : "m" (v->counter)
                          : "memory", "cc"
                          : exit);
        fail_fn(v);
exit:
        return;
}
```

Actually, there is only one difference. We increment value if the `mutex->count`. So it will represent `unlocked` state after this operation. As `mutex` released, but we have something in the `wait queue` we need to update it. In this case the `fail_fn` function will be called which is `__mutex_unlock_slowpath`. The `__mutex_unlock_slowpath` function just gets the correct `mutex` instance by the given `mutex->count` and calls the `__mutex_unlock_common_slowpath` function:

```
__mutex_unlock_slowpath(atomic_t *lock_count)
{
      struct mutex *lock = container_of(lock_count, struct mutex, count);

      __mutex_unlock_common_slowpath(lock, 1);
}
```

In the `__mutex_unlock_common_slowpath` function we will get the first entry from the wait queue if the wait queue is not empty and wakeup related process:

```
if (!list_empty(&lock->wait_list)) {
    struct mutex_waiter *waiter =
           list_entry(lock->wait_list.next, struct mutex_waiter, list);
                wake_up_process(waiter->task);
}
```

After this, a mutex will be released by previous process and will be acquired by another process from a wait queue.

That's all. We have considered main `API` for manipulation with `mutexes`: `mutex_lock` and `mutex_unlock`. Besides this the Linux kernel provides following API:

- `mutex_lock_interruptible`;
- `mutex_lock_killable`;
- `mutex_trylock`.

and corresponding versions of `unlock` prefixed functions. This part will not describe this `API`, because it is similar to corresponding `API` of `semaphores`. More about it you may read in the previous part.

That's all.

Conclusion

This is the end of the fourth part of the synchronization primitives chapter in the Linux kernel. In this part we met with new synchronization primitive which is called - `mutex`. From the theoretical side, this synchronization primitive very similar on a semaphore. Actually, `mutex` represents binary semaphore. But its implementation differs from the implementation of `semaphore` in the Linux kernel. In the next part we will continue to dive into synchronization primitives in the Linux kernel.

If you have questions or suggestions, feel free to ping me in twitter 0xAX, drop me email or just create issue.

Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-insides.

Links

- Mutex
- Spinlock
- Semaphore
- Synchronization primitives
- API
- Locking mechanism
- Context switches
- lock validator
- Atomic
- MCS lock
- Doubly linked list
- x86_64
- Inline assembly
- Memory barrier
- Lock instruction
- JNS instruction
- preemtion

- Unix signals
- Previous part

Synchronization primitives in the Linux kernel. Part 5.

Introduction

This is the fifth part of the chapter which describes synchronization primitives in the Linux kernel and in the previous parts we finished to consider different types spinlocks, semaphore and mutex synchronization primitives. We will continue to learn synchronization primitives in this part and start to consider special type of synchronization primitives - readers–writer lock.

The first synchronization primitive of this type will be already familiar for us - semaphore. As in all previous parts of this book, before we will consider implementation of the `reader/writer semaphores` in the Linux kernel, we will start from the theoretical side and will try to understand what is the difference between `reader/writer semaphores` and `normal semaphores`.

So, let's start.

Reader/Writer semaphore

Actually there are two types of operations may be performed on the data. We may read data and make changes in data. Two fundamental operations - `read` and `write`. Usually (but not always), `read` operation is performed more often than `write` operation. In this case, it would be logical to we may lock data in such way, that some processes may read locked data in one time, on condition that no one will not change the data. The readers/writer lock allows us to get this lock.

When a process which wants to write something into data, all other `writer` and `reader` processes will be blocked until the process which acquired a lock, will not release it. When a process reads data, other processes which want to read the same data too, will not be locked and will be able to do this. As you may guess, implementation of the `reader/writer semaphore` is based on the implementation of the `normal semaphore`. We already familiar with the semaphore synchronization primitive from the third part of this chapter. From the theoretical side everything looks pretty simple. Let's look how `reader/writer semaphore` is represented in the Linux kernel.

The `semaphore` is represented by the:

```
struct semaphore {
    raw_spinlock_t        lock;
    unsigned int        count;
    struct list_head    wait_list;
};
```

structure. If you will look in the include/linux/rwsem.h header file, you will find definition of the `rw_semaphore` structure which represents `reader/writer semaphore` in the Linux kernel. Let's look at the definition of this structure:

```
#ifdef CONFIG_RWSEM_GENERIC_SPINLOCK
#include <linux/rwsem-spinlock.h>
#else
struct rw_semaphore {
        long count;
        struct list_head wait_list;
        raw_spinlock_t wait_lock;
#ifdef CONFIG_RWSEM_SPIN_ON_OWNER
        struct optimistic_spin_queue osq;
        struct task_struct *owner;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
        struct lockdep_map      dep_map;
#endif
};
```

Before we will consider fields of the `rw_semaphore` structure, we may notice, that declaration of the `rw_semaphore` structure depends on the `CONFIG_RWSEM_GENERIC_SPINLOCK` kernel configuration option. This option is disabled for the x86_64 architecture by default. We can be sure in this by looking at the corresponding kernel configuration file. In our case, this configuration file is - arch/x86/um/Kconfig:

```
config RWSEM_XCHGADD_ALGORITHM
    def_bool 64BIT

config RWSEM_GENERIC_SPINLOCK
    def_bool !RWSEM_XCHGADD_ALGORITHM
```

So, as this book describes only x86_64 architecture related stuff, we will skip the case when the `CONFIG_RWSEM_GENERIC_SPINLOCK` kernel configuration is enabled and consider definition of the `rw_semaphore` structure only from the include/linux/rwsem.h header file.

If we will take a look at the definition of the `rw_semaphore` structure, we will notice that first three fields are the same that in the `semaphore` structure. It contains `count` field which represents amount of available resources, the `wait_list` field which represents doubly

linked list of processes which are waiting to acquire a lock and `wait_lock` spinlock for protection of this list. Notice that `rw_semaphore.count` field is `long` type unlike the same field in the `semaphore` structure.

The `count` field of a `rw_semaphore` structure may have following values:

- `0x0000000000000000` - `reader/writer semaphore` is in unlocked state and no one is waiting for a lock;
- `0x000000000000000X` - `X` readers are active or attempting to acquire a lock and no writer waiting;
- `0xffffffff0000000X` - may represent different cases. The first is - `X` readers are active or attempting to acquire a lock with waiters for the lock. The second is - one writer attempting a lock, no waiters for the lock. And the last - one writer is active and no waiters for the lock;
- `0xffffffff00000001` - may represented two different cases. The first is - one reader is active or attempting to acquire a lock and exist waiters for the lock. The second case is one writer is active or attempting to acquire a lock and no waiters for the lock;
- `0xffffffff00000000` - represents situation when there are readers or writers are queued, but no one is active or is in the process of acquire of a lock;
- `0xfffffffe00000001` - a writer is active or attempting to acquire a lock and waiters are in queue.

So, besides the `count` field, all of these fields are similar to fields of the `semaphore` structure. Last three fields depend on the two configuration options of the Linux kernel: the `CONFIG_RWSEM_SPIN_ON_OWNER` and `CONFIG_DEBUG_LOCK_ALLOC`. The first two fields may be familiar us by declaration of the mutex structure from the previous part. The first `osq` field represents MCS lock spinner for `optimistic spinning` and the second represents process which is current owner of a lock.

The last field of the `rw_semaphore` structure is - `dep_map` - debugging related, and as I already wrote in previous parts, we will skip debugging related stuff in this chapter.

That's all. Now we know a little about what is it `reader/writer lock` in general and `reader/writer semaphore` in particular. Additionally we saw how a `reader/writer semaphore` is represented in the Linux kernel. In this case, we may go ahead and start to look at the API which the Linux kernel provides for manipulation of `reader/writer semaphores`.

Reader/Writer semaphore API

So, we know a little about `reader/writer semaphores` from theoretical side, let's look on its implementation in the Linux kernel. All `reader/writer semaphores` related API is located in the include/linux/rwsem.h header file.

As always Before we will consider an API of the `reader/writer semaphore` mechanism in the Linux kernel, we need to know how to initialize the `rw_semaphore` structure. As we already saw in previous parts of this chapter, all synchronization primitives may be initialized in two ways:

- `statically`;
- `dynamically`.

And `reader/writer semaphore` is not an exception. First of all, let's take a look at the first approach. We may initialize `rw_semaphore` structure with the help of the `DECLARE_RWSEM` macro in compile time. This macro is defined in the include/linux/rwsem.h header file and looks:

```
#define DECLARE_RWSEM(name) \
        struct rw_semaphore name = __RWSEM_INITIALIZER(name)
```

As we may see, the `DECLARE_RWSEM` macro just expands to the definition of the `rw_semaphore` structure with the given name. Additionally new `rw_semaphore` structure is initialized with the value of the `__RWSEM_INITIALIZER` macro:

```
#define __RWSEM_INITIALIZER(name)              \
{                                                               \
        .count = RWSEM_UNLOCKED_VALUE,                          \
        .wait_list = LIST_HEAD_INIT((name).wait_list),          \
        .wait_lock = __RAW_SPIN_LOCK_UNLOCKED(name.wait_lock)   \
         __RWSEM_OPT_INIT(name)                                 \
         __RWSEM_DEP_MAP_INIT(name)
}
```

and expands to the initialization of fields of `rw_semaphore` structure. First of all we initialize `count` field of the `rw_semaphore` structure to the `unlocked` state with `RWSEM_UNLOCKED_VALUE` macro from the arch/x86/include/asm/rwsem.h architecture specific header file:

```
#define RWSEM_UNLOCKED_VALUE            0x00000000L
```

After this we initialize list of a lock waiters with the empty linked list and spinlock for protection of this list with the `unlocked` state too. The `__RWSEM_OPT_INIT` macro depends on the state of the `CONFIG_RWSEM_SPIN_ON_OWNER` kernel configuration option and if this option is enabled it expands to the initialization of the `osq` and `owner` fields of the `rw_semaphore` structure. As we already saw above, the `CONFIG_RWSEM_SPIN_ON_OWNER` kernel configuration option is enabled by default for x86_64 architecture, so let's take a look at the definition of the `__RWSEM_OPT_INIT` macro:

```
#ifdef CONFIG_RWSEM_SPIN_ON_OWNER
    #define __RWSEM_OPT_INIT(lockname) , .osq = OSQ_LOCK_UNLOCKED, .owner = NULL
#else
    #define __RWSEM_OPT_INIT(lockname)
#endif
```

As we may see, the `__RWSEM_OPT_INIT` macro initializes the MCS lock lock with `unlocked` state and initial `owner` of a lock with `NULL`. From this moment, a `rw_semaphore` structure will be initialized in a compile time and may be used for data protection.

The second way to initialize a `rw_semaphore` structure is `dynamically` or use the `init_rwsem` macro from the include/linux/rwsem.h header file. This macro declares an instance of the `lock_class_key` which is related to the lock validator of the Linux kernel and to the call of the `__init_rwsem` function with the given `reader/writer semaphore`:

```
#define init_rwsem(sem)                          \
do {                                                            \
        static struct lock_class_key __key;                     \
                                                                \
        __init_rwsem((sem), #sem, &__key);                      \
} while (0)
```

If you will start definition of the `__init_rwsem` function, you will notice that there are couple of source code files which contain it. As you may guess, sometimes we need to initialize additional fields of the `rw_semaphore` structure, like the `osq` and `owner`. But sometimes not. All of this depends on some kernel configuration options. If we will look at the kernel/locking/Makefile makefile, we will see following lines:

```
obj-$(CONFIG_RWSEM_GENERIC_SPINLOCK) += rwsem-spinlock.o
obj-$(CONFIG_RWSEM_XCHGADD_ALGORITHM) += rwsem-xadd.o
```

As we already know, the Linux kernel for `x86_64` architecture has enabled `CONFIG_RWSEM_XCHGADD_ALGORITHM` kernel configuration option by default:

```
config RWSEM_XCHGADD_ALGORITHM
    def_bool 64BIT
```

in the arch/x86/um/Kconfig kernel configuration file. In this case, implementation of the `__init_rwsem` function will be located in the kernel/locking/rwsem-xadd.c source code file for us. Let's take a look at this function:

```
void __init_rwsem(struct rw_semaphore *sem, const char *name,
                  struct lock_class_key *key)
{
#ifdef CONFIG_DEBUG_LOCK_ALLOC
        debug_check_no_locks_freed((void *)sem, sizeof(*sem));
        lockdep_init_map(&sem->dep_map, name, key, 0);
#endif
        sem->count = RWSEM_UNLOCKED_VALUE;
        raw_spin_lock_init(&sem->wait_lock);
        INIT_LIST_HEAD(&sem->wait_list);
#ifdef CONFIG_RWSEM_SPIN_ON_OWNER
        sem->owner = NULL;
        osq_lock_init(&sem->osq);
#endif
}
```

We may see here almost the same as in `__RWSEM_INITIALIZER` macro with difference that all of this will be executed in runtime.

So, from now we are able to initialize a `reader/writer semaphore` let's look at the `lock` and `unlock` API. The Linux kernel provides following primary API to manipulate `reader/writer semaphores` :

- `void down_read(struct rw_semaphore *sem)` - lock for reading;
- `int down_read_trylock(struct rw_semaphore *sem)` - try lock for reading;
- `void down_write(struct rw_semaphore *sem)` - lock for writing;
- `int down_write_trylock(struct rw_semaphore *sem)` - try lock for writing;
- `void up_read(struct rw_semaphore *sem)` - release a read lock;
- `void up_write(struct rw_semaphore *sem)` - release a write lock;

Let's start as always from the locking. First of all let's consider implementation of the `down_write` function which executes a try of acquiring of a lock for `write` . This function is kernel/locking/rwsem.c source code file and starts from the call of the macro from the include/linux/kernel.h header file:

```
void __sched down_write(struct rw_semaphore *sem)
{
        might_sleep();
        rwsem_acquire(&sem->dep_map, 0, 0, _RET_IP_);

        LOCK_CONTENDED(sem, __down_write_trylock, __down_write);
        rwsem_set_owner(sem);
}
```

We already met the `might_sleep` macro in the previous part. In short words, Implementation of the `might_sleep` macro depends on the `CONFIG_DEBUG_ATOMIC_SLEEP` kernel configuration option and if this option is enabled, this macro just prints a stack trace if it was executed in atomic context. As this macro is mostly for debugging purpose we will skip it and will go ahead. Additionally we will skip the next macro from the `down_read` function - `rwsem_acquire` which is related to the lock validator of the Linux kernel, because this is topic of other part.

The only two things that remained in the `down_write` function is the call of the `LOCK_CONTENDED` macro which is defined in the include/linux/lockdep.h header file and setting of owner of a lock with the `rwsem_set_owner` function which sets owner to currently running process:

```
static inline void rwsem_set_owner(struct rw_semaphore *sem)
{
        sem->owner = current;
}
```

As you already may guess, the `LOCK_CONTENDED` macro does all job for us. Let's look at the implementation of the `LOCK_CONTENDED` macro:

```
#define LOCK_CONTENDED(_lock, try, lock) \
        lock(_lock)
```

As we may see it just calls the `lock` function which is third parameter of the `LOCK_CONTENDED` macro with the given `rw_semaphore`. In our case the third parameter of the `LOCK_CONTENDED` macro is the `__down_write` function which is architecture specific function and located in the arch/x86/include/asm/rwsem.h header file. Let's look at the implementation of the `__down_write` function:

```
static inline void __down_write(struct rw_semaphore *sem)
{
        __down_write_nested(sem, 0);
}
```

which just executes a call of the `__down_write_nested` function from the same source code file. Let's take a look at the implementation of the `__down_write_nested` function:

```
static inline void __down_write_nested(struct rw_semaphore *sem, int subclass)
{
        long tmp;

        asm volatile("# beginning down_write\n\t"
                     LOCK_PREFIX "  xadd      %1,(%2)\n\t"
                     "  test " __ASM_SEL(%w1,%k1) "," __ASM_SEL(%w1,%k1) "\n\t"
                     "  jz        1f\n"
                     "  call call_rwsem_down_write_failed\n"
                     "1:\n"
                     "# ending down_write"
                     : "+m" (sem->count), "=d" (tmp)
                     : "a" (sem), "1" (RWSEM_ACTIVE_WRITE_BIAS)
                     : "memory", "cc");
}
```

As for other synchronization primitives which we saw in this chapter, usually `lock/unlock` functions consists only from an inline assembly statement. As we may see, in our case the same for `__down_write_nested` function. Let's try to understand what does this function do. The first line of our assembly statement is just a comment, let's skip it. The second like contains `LOCK_PREFIX` which will be expanded to the LOCK instruction as we already know. The next xadd instruction executes `add` and `exchange` operations. In other words, `xadd` instruction adds value of the `RWSEM_ACTIVE_WRITE_BIAS` :

```
#define RWSEM_ACTIVE_WRITE_BIAS         (RWSEM_WAITING_BIAS + RWSEM_ACTIVE_BIAS)

#define RWSEM_WAITING_BIAS              (-RWSEM_ACTIVE_MASK-1)
#define RWSEM_ACTIVE_BIAS               0x00000001L
```

or `0xffffffff00000001` to the `count` of the given `reader/writer semaphore` and returns previous value of it. After this we check the active mask in the `rw_semaphore->count` . If it was zero before, this means that there were no-one writer before, so we acquired a lock. In other way we call the `call_rwsem_down_write_failed` function from the arch/x86/lib/rwsem.S assembly file. The the `call_rwsem_down_write_failed` function just calls the `rwsem_down_write_failed` function from the kernel/locking/rwsem-xadd.c source code file anticipatorily save general purpose registers:

```
ENTRY(call_rwsem_down_write_failed)
    FRAME_BEGIN
    save_common_regs
    movq %rax,%rdi
    call rwsem_down_write_failed
    restore_common_regs
    FRAME_END
    ret
    ENDPROC(call_rwsem_down_write_failed)
```

The `rwsem_down_write_failed` function starts from the atomic update of the `count` value:

```
 __visible
struct rw_semaphore __sched *rwsem_down_write_failed(struct rw_semaphore *sem)
{
    count = rwsem_atomic_update(-RWSEM_ACTIVE_WRITE_BIAS, sem);
    ...
    ...
    ...
}
```

with the `-RWSEM_ACTIVE_WRITE_BIAS` value. The `rwsem_atomic_update` function is defined in the arch/x86/include/asm/rwsem.h header file and implement exchange and add logic:

```
static inline long rwsem_atomic_update(long delta, struct rw_semaphore *sem)
{
        return delta + xadd(&sem->count, delta);
}
```

This function atomically adds the given delta to the `count` and returns old value of the count. After this it just returns sum of the given `delta` and old value of the `count` field. In our case we undo write bias from the `count` as we didn't acquire a lock. After this step we try to do `optimistic spinning` by the call of the `rwsem_optimistic_spin` function:

```
if (rwsem_optimistic_spin(sem))
      return sem;
```

We will skip implementation of the `rwsem_optimistic_spin` function, as it is similar on the `mutex_optimistic_spin` function which we saw in the previous part. In short words we check existence other tasks ready to run that have higher priority in the `rwsem_optimistic_spin` function. If there are such tasks, the process will be added to the MCS `waitqueue` and start to spin in the loop until a lock will be able to be acquired. If `optimistic spinning` is disabled, a process will be added to the and marked as waiting for write:

```
waiter.task = current;
waiter.type = RWSEM_WAITING_FOR_WRITE;

if (list_empty(&sem->wait_list))
    waiting = false;

list_add_tail(&waiter.list, &sem->wait_list);
```

waiters list and start to wait until it will successfully acquire the lock. After we have added a process to the waiters list which was empty before this moment, we update the value of the `rw_semaphore->count` with the `RWSEM_WAITING_BIAS` :

```
count = rwsem_atomic_update(RWSEM_WAITING_BIAS, sem);
```

with this we mark `rw_semaphore->counter` that it is already locked and exists/waits one `writer` which wants to acquire the lock. In other way we try to wake `reader` processes from the `wait queue` that were queued before this `writer` process and there are no active readers. In the end of the `rwsem_down_write_failed` a `writer` process will go to sleep which didn't acquire a lock in the following loop:

```
while (true) {
    if (rwsem_try_write_lock(count, sem))
        break;
    raw_spin_unlock_irq(&sem->wait_lock);
    do {
        schedule();
        set_current_state(TASK_UNINTERRUPTIBLE);
    } while ((count = sem->count) & RWSEM_ACTIVE_MASK);
    raw_spin_lock_irq(&sem->wait_lock);
}
```

I will skip explanation of this loop as we already met similar functional in the previous part.

That's all. From this moment, our `writer` process will acquire or not acquire a lock depends on the value of the `rw_semaphore->count` field. Now if we will look at the implementation of the `down_read` function which executes a try of acquiring of a lock. We will see similar actions which we saw in the `down_write` function. This function calls different debugging and lock validator related functions/macros:

```
void __sched down_read(struct rw_semaphore *sem)
{
        might_sleep();
        rwsem_acquire_read(&sem->dep_map, 0, 0, _RET_IP_);

        LOCK_CONTENDED(sem, __down_read_trylock, __down_read);
}
```

and does all job in the `__down_read` function. The `__down_read` consists of inline assembly statement:

```
static inline void __down_read(struct rw_semaphore *sem)
{
         asm volatile("# beginning down_read\n\t"
                      LOCK_PREFIX _ASM_INC "(%1)\n\t"
                      "  jns        1f\n"
                      "  call call_rwsem_down_read_failed\n"
                      "1:\n\t"
                      "# ending down_read\n\t"
                      : "+m" (sem->count)
                      : "a" (sem)
                      : "memory", "cc");
}
```

which increments value of the given `rw_semaphore->count` and call the `call_rwsem_down_read_failed` if this value is negative. In other way we jump at the label `1:` and exit. After this `read` lock will be successfully acquired. Notice that we check a sign of the `count` value as it may be negative, because as you may remember most significant word of the `rw_semaphore->count` contains negated number of active writers.

Let's consider case when a process wants to acquire a lock for `read` operation, but it is already locked. In this case the `call_rwsem_down_read_failed` function from the arch/x86/lib/rwsem.S assembly file will be called. If you will look at the implementation of this function, you will notice that it does the same that `call_rwsem_down_read_failed` function does. Except it calls the `rwsem_down_read_failed` function instead of `rwsem_dow_write_failed`. Now let's consider implementation of the `rwsem_down_read_failed` function. It starts from the adding a process to the `wait queue` and updating of value of the `rw_semaphore->counter`:

```
long adjustment = -RWSEM_ACTIVE_READ_BIAS;

waiter.task = tsk;
waiter.type = RWSEM_WAITING_FOR_READ;

if (list_empty(&sem->wait_list))
    adjustment += RWSEM_WAITING_BIAS;
list_add_tail(&waiter.list, &sem->wait_list);

count = rwsem_atomic_update(adjustment, sem);
```

Notice that if the `wait queue` was empty before we clear the `rw_semaphore->counter` and undo `read` bias in other way. At the next step we check that there are no active locks and we are first in the `wait queue` we need to join currently active `reader` processes. In other way we go to sleep until a lock will not be able to acquired.

That's all. Now we know how `reader` and `writer` processes will behave in different cases during a lock acquisition. Now let's take a short look at `unlock` operations. The `up_read` and `up_write` functions allows us to unlock a `reader` or `writer` lock. First of all let's take a look at the implementation of the `up_write` function which is defined in the kernel/locking/rwsem.c source code file:

```
void up_write(struct rw_semaphore *sem)
{
        rwsem_release(&sem->dep_map, 1, _RET_IP_);

        rwsem_clear_owner(sem);
        __up_write(sem);
}
```

First of all it calls the `rwsem_release` macro which is related to the lock validator of the Linux kernel, so we will skip it now. And at the next line the `rwsem_clear_owner` function which as you may understand from the name of this function, just clears the `owner` field of the given `rw_semaphore`:

```
static inline void rwsem_clear_owner(struct rw_semaphore *sem)
{
    sem->owner = NULL;
}
```

The `__up_write` function does all job of unlocking of the lock. The `_up_write` is architecture-specific function, so for our case it will be located in the arch/x86/include/asm/rwsem.h source code file. If we will take a look at the implementation

of this function, we will see that it does almost the same that `__down_write` function, but conversely. Instead of adding of the `RWSEM_ACTIVE_WRITE_BIAS` to the `count` , we subtract the same value and check the `sign` of the previous value.

If the previous value of the `rw_semaphore->count` is not negative, a writer process released a lock and now it may be acquired by someone else. In other case, the `rw_semaphore->count` will contain negative values. This means that there is at least one `writer` in a wait queue. In this case the `call_rwsem_wake` function will be called. This function acts like similar functions which we already saw above. It store general purpose registers at the stack for preserving and call the `rwsem_wake` function.

First of all the `rwsem_wake` function checks if a spinner is present. In this case it will just acquire a lock which is just released by lock owner. In other case there must be someone in the `wait queue` and we need to wake or writer process if it exists at the top of the `wait queue` or all `reader` processes. The `up_read` function which release a `reader` lock acts in similar way like `up_write` , but with a little difference. Instead of subtracting of `RWSEM_ACTIVE_WRITE_BIAS` from the `rw_semaphore->count` , it subtracts `1` from it, because less significant word of the `count` contains number active locks. After this it checks `sign` of the `count` and calls the `rwsem_wake` like `__up_write` if the `count` is negative or in other way lock will be successfully released.

That's all. We have considered API for manipulation with `reader/writer semaphore` : `up_read/up_write` and `down_read/down_write` . We saw that the Linux kernel provides additional API, besides this functions, like the `,` and etc. But I will not consider implementation of these function in this part because it must be similar on that we have seen in this part of except few subtleties.

Conclusion

This is the end of the fifth part of the synchronization primitives chapter in the Linux kernel. In this part we met with special type of `semaphore` - `readers/writer` semaphore which provides access to data for multiply process to read or for one process to writer. In the next part we will continue to dive into synchronization primitives in the Linux kernel.

If you have questions or suggestions, feel free to ping me in twitter 0xAX, drop me email or just create issue.

Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-insides.

Links

- Synchronization primitives
- Readers/Writer lock
- Spinlocks
- Semaphore
- Mutex
- x86_64 architecture
- Doubly linked list
- MCS lock
- API
- Linux kernel lock validator
- Atomic operations
- Inline assembly
- XADD instruction
- LOCK instruction
- Previous part

Synchronization primitives in the Linux kernel. Part 6.

Introduction

This is the sixth part of the chapter which describes synchronization primitives) in the Linux kernel and in the previous parts we finished to consider different readers-writer lock synchronization primitives. We will continue to learn synchronization primitives in this part and start to consider a similar synchronization primitive which can be used to avoid the `writer starvation` problem. The name of this synchronization primitive is - `seqlock` or `sequential locks`.

We know from the previous part that readers-writer lock is a special lock mechanism which allows concurrent access for read-only operations, but an exclusive lock is needed for writing or modifying data. As we may guess, it may lead to a problem which is called `writer starvation`. In other words, a writer process can't acquire a lock as long as at least one reader process which aqcuired a lock holds it. So, in the situation when contention is high, it will lead to situation when a writer process which wants to acquire a lock will wait for it for a long time.

The `seqlock` synchronization primitive can help solve this problem.

As in all previous parts of this book, we will try to consider this synchronization primitive from the theoretical side and only than we will consider API provided by the Linux kernel to manipulate with `seqlocks`.

So, let's start.

Sequential lock

So, what is a `seqlock` synchronization primitive and how does it work? Let's try to answer on these questions in this paragraph. Actually `sequential locks` were introduced in the Linux kernel 2.6.x. Main point of this synchronization primitive is to provide fast and lock-free access to shared resources. Since the heart of `sequential lock` synchronization primitive is spinlock synchronization primitive, `sequential locks` work in situations where the protected resources are small and simple. Additionally write access must be rare and also should be fast.

Work of this synchronization primitive is based on the sequence of events counter. Actually a `sequential lock` allows free access to a resource for readers, but each reader must check existence of conflicts with a writer. This synchronization primitive introduces a special counter. The main algorithm of work of `sequential locks` is simple: Each writer which acquired a sequential lock increments this counter and additionally acquires a spinlock. When this writer finishes, it will release the acquired spinlock to give access to other writers and increment the counter of a sequential lock again.

Read only access works on the following principle, it gets the value of a `sequential lock` counter before it will enter into critical section and compares it with the value of the same `sequential lock` counter at the exit of critical section. If their values are equal, this means that there weren't writers for this period. If their values are not equal, this means that a writer has incremented the counter during the critical section. This conflict means that reading of protected data must be repeated.

That's all. As we may see principle of work of `sequential locks` is simple.

```
unsigned int seq_counter_value;

do {
    seq_counter_value = get_seq_counter_val(&the_lock);
    //
    // do as we want here
    //
} while (__retry__);
```

Actually the Linux kernel does not provide `get_seq_counter_val()` function. Here it is just a stub. Like a `__retry__` too. As I already wrote above, we will see actual the API for this in the next paragraph of this part.

Ok, now we know what a `seqlock` synchronization primitive is and how it is represented in the Linux kernel. In this case, we may go ahead and start to look at the API which the Linux kernel provides for manipulation of synchronization primitives of this type.

Sequential lock API

So, now we know a little about `sequentional lock` synchronization primitive from theoretical side, let's look at its implementation in the Linux kernel. All `sequentional locks` API are located in the include/linux/seqlock.h header file.

First of all we may see that the a `sequential lock` machanism is represented by the following type:

```
typedef struct {
	struct seqcount seqcount;
	spinlock_t lock;
} seqlock_t;
```

As we may see the `seqlock_t` provides two fields. These fields represent a sequential lock counter, description of which we saw above and also a spinlock which will protect data from other writers. Note that the `seqcount` counter represented as `seqcount` type. The `seqcount` is structure:

```
typedef struct seqcount {
	unsigned sequence;
#ifdef CONFIG_DEBUG_LOCK_ALLOC
	struct lockdep_map dep_map;
#endif
} seqcount_t;
```

which holds counter of a sequential lock and lock validator related field.

As always in previous parts of this chapter, before we will consider an API of `sequential lock` mechanism in the Linux kernel, we need to know how to initialize an instance of `seqlock_t`.

We saw in the previous parts that often the Linux kernel provides two approaches to execute initialization of the given synchronization primitive. The same situation with the `seqlock_t` structure. These approaches allows to initialize a `seqlock_t` in two following:

- `statically`;
- `dynamically`.

ways. Let's look at the first approach. We are able to intialize a `seqlock_t` statically with the `DEFINE_SEQLOCK` macro:

```
#define DEFINE_SEQLOCK(x) \
		seqlock_t x = __SEQLOCK_UNLOCKED(x)
```

which is defined in the include/linux/seqlock.h header file. As we may see, the `DEFINE_SEQLOCK` macro takes one argument and expands to the definition and initialization of the `seqlock_t` structure. Initialization occurs with the help of the `__SEQLOCK_UNLOCKED` macro which is defined in the same source code file. Let's look at the implementation of this macro:

```
#define __SEQLOCK_UNLOCKED(lockname)            \
    {                       \
        .seqcount = SEQCNT_ZERO(lockname),    \
        .lock =    __SPIN_LOCK_UNLOCKED(lockname)    \
    }
```

As we may see the, `__SEQLOCK_UNLOCKED` macro executes initialization of fields of the given `seqlock_t` structure. The first field is `seqcount` initialized with the `SEQCNT_ZERO` macro which expands to the:

```
#define SEQCNT_ZERO(lockname) { .sequence = 0, SEQCOUNT_DEP_MAP_INIT(lockname)}
```

So we just initialize counter of the given sequential lock to zero and additionally we can see lock validator related initialization which depends on the state of the `CONFIG_DEBUG_LOCK_ALLOC` kernel configuration option:

```
#ifdef CONFIG_DEBUG_LOCK_ALLOC
# define SEQCOUNT_DEP_MAP_INIT(lockname) \
    .dep_map = { .name = #lockname } \
    ...
    ...
    ...
#else
# define SEQCOUNT_DEP_MAP_INIT(lockname)
    ...
    ...
    ...
#endif
```

As I already wrote in previous parts of this chapter we will not consider debugging and lock validator related stuff in this part. So for now we just skip the `SEQCOUNT_DEP_MAP_INIT` macro. The second field of the given `seqlock_t` is `lock` initialized with the `__SPIN_LOCK_UNLOCKED` macro which is defined in the include/linux/spinlock_types.h header file. We will not consider implementation of this macro here as it just initialize rawspinlock with architecture-specific methods (More abot spinlocks you may read in first parts of this chapter).

We have considered the first way to initialize a sequential lock. Let's consider second way to do the same, but do it dynamically. We can initialize a sequentional lock with the `seqlock_init` macro which is defined in the same include/linux/seqlock.h header file.

Let's look at the implementation of this macro:

```
#define seqlock_init(x)                    \
    do {                           \
        seqcount_init(&(x)->seqcount);        \
        spin_lock_init(&(x)->lock);        \
    } while (0)
```

As we may see, the `seqlock_init` expands into two macros. The first macro `seqcount_init` takes counter of the given sequential lock and expands to the call of the `__seqcount_init` function:

```
# define seqcount_init(s)                 \
    do {                           \
        static struct lock_class_key __key;    \
        __seqcount_init((s), #s, &__key);    \
    } while (0)
```

from the same header file. This function

```
static inline void __seqcount_init(seqcount_t *s, const char *name,
                      struct lock_class_key *key)
{
    lockdep_init_map(&s->dep_map, name, key, 0);
    s->sequence = 0;
}
```

just initializes counter of the given `seqcount_t` with zero. The second call from the `seqlock_init` macro is the call of the `spin_lock_init` macro which we saw in the first part of this chapter.

So, now we know how to initialize a `sequential lock`, now let's look at how to use it. The Linux kernel provides following API to manipulate `sequential locks`:

```
static inline unsigned read_seqbegin(const seqlock_t *sl);
static inline unsigned read_seqretry(const seqlock_t *sl, unsigned start);
static inline void write_seqlock(seqlock_t *sl);
static inline void write_sequnlock(seqlock_t *sl);
static inline void write_seqlock_irq(seqlock_t *sl);
static inline void write_sequnlock_irq(seqlock_t *sl);
static inline void read_seqlock_excl(seqlock_t *sl)
static inline void read_sequnlock_excl(seqlock_t *sl)
```

and others. Before we move on to considering the implementation of this API, we must know that actually there are two types of readers. The first type of reader never blocks a writer process. In this case writer will not wait for readers. The second type of reader which can

lock. In this case, the locking reader will block the writer as it will wait while reader will not release its lock.

First of all let's consider the first type of readers. The `read_seqbegin` function begins a seq-read critical section.

As we may see this function just returns value of the `read_seqcount_begin` function:

```
static inline unsigned read_seqbegin(const seqlock_t *sl)
{
    return read_seqcount_begin(&sl->seqcount);
}
```

In its turn the `read_seqcount_begin` function calls the `raw_read_seqcount_begin` function:

```
static inline unsigned read_seqcount_begin(const seqcount_t *s)
{
    return raw_read_seqcount_begin(s);
}
```

which just returns value of the `sequential lock` counter:

```
static inline unsigned raw_read_seqcount(const seqcount_t *s)
{
    unsigned ret = READ_ONCE(s->sequence);
    smp_rmb();
    return ret;
}
```

After we have the initial value of the given `sequential lock` counter and did some stuff, we know from the previous paragraph of this function, that we need to compare it with the current value of the counter the same `sequential lock` before we will exit from the critical section. We can achieve this by the call of the `read_seqretry` function. This function takes a `sequential lock`, start value of the counter and through a chain of functions:

```
static inline unsigned read_seqretry(const seqlock_t *sl, unsigned start)
{
    return read_seqcount_retry(&sl->seqcount, start);
}

static inline int read_seqcount_retry(const seqcount_t *s, unsigned start)
{
    smp_rmb();
    return __read_seqcount_retry(s, start);
}
```

it calls the `__read_seqcount_retry` function:

```
static inline int __read_seqcount_retry(const seqcount_t *s, unsigned start)
{
    return unlikely(s->sequence != start);
}
```

which just compares value of the counter of the given `sequential lock` with the initial value of this counter. If the initial value of the counter which is obtained from `read_seqbegin()` function is odd, this means that a writer was in the middle of updating the data when our reader began to act. In this case the value of the data can be in inconsistent state, so we need to try to read it again.

This is a common pattern in the Linux kernel. For example, you may remember the `jiffies` concept from the first part of the timers and time management in the Linux kernel chapter. The sequential lock is used to obtain value of `jiffies` at x86_64 architecture:

```
u64 get_jiffies_64(void)
{
    unsigned long seq;
    u64 ret;

    do {
        seq = read_seqbegin(&jiffies_lock);
        ret = jiffies_64;
    } while (read_seqretry(&jiffies_lock, seq));
    return ret;
}
```

Here we just read the value of the counter of the `jiffies_lock` sequential lock and then we write value of the `jiffies_64` system variable to the `ret`. As here we may see `do/while` loop, the body of the loop will be executed at least one time. So, as the body of loop was executed, we read and compare the current value of the counter of the `jiffies_lock` with the initial value. If these values are not equal, execution of the loop will be repeated, else `get_jiffies_64` will return its value in `ret`.

We just saw the first type of readers which do not block writer and other readers. Let's consider second type. It does not update value of a `sequential lock` counter, but just locks `spinlock`:

```
static inline void read_seqlock_excl(seqlock_t *sl)
{
    spin_lock(&sl->lock);
}
```

So, no one reader or writer can't access protected data. When a reader finishes, the lock must be unlocked with the:

```
static inline void read_sequnlock_excl(seqlock_t *sl)
{
    spin_unlock(&sl->lock);
}
```

function.

Now we know how `sequential lock` work for readers. Let's consider how does writer act when it wants to acquire a `sequential lock` to modify data. To acquire a `sequential lock`, writer should use `write_seqlock` function. If we look at the implementation of this function:

```
static inline void write_seqlock(seqlock_t *sl)
{
    spin_lock(&sl->lock);
    write_seqcount_begin(&sl->seqcount);
}
```

We will see that it acquires `spinlock` to prevent access from other writers and calls the `write_seqcount_begin` function. This function just increments value of the `sequential lock` counter:

```
static inline void raw_write_seqcount_begin(seqcount_t *s)
{
    s->sequence++;
    smp_wmb();
}
```

When a writer process will finish to modify data, the `write_sequnlock` function must be called to release a lock and give access to other writers or readers. Let's consider at the implementation of the `write_sequnlock` function. It looks pretty simple:

```
static inline void write_sequnlock(seqlock_t *sl)
{
    write_seqcount_end(&sl->seqcount);
    spin_unlock(&sl->lock);
}
```

First of all it just calls `write_seqcount_end` function to increase value of the counter of the `sequential` lock again:

```
static inline void raw_write_seqcount_end(seqcount_t *s)
{
    smp_wmb();
    s->sequence++;
}
```

and in the end we just call the `spin_unlock` macro to give access for other readers or writers.

That's all about `sequential lock` mechanism in the Linux kernel. Of course we did not consider full API of this mechanism in this part. But all other functions are based on these which we described here. For example, Linux kernel also provides some safe macros/functions to use `sequential lock` mechanism in interrupt handlers of softirq: `write_seqclock_irq` and `write_sequnlock_irq` :

```
static inline void write_seqlock_irq(seqlock_t *sl)
{
    spin_lock_irq(&sl->lock);
    write_seqcount_begin(&sl->seqcount);
}

static inline void write_sequnlock_irq(seqlock_t *sl)
{
    write_seqcount_end(&sl->seqcount);
    spin_unlock_irq(&sl->lock);
}
```

As we may see, these functions differ only in the initialization of spinlock. They call `spin_lock_irq` and `spin_unlock_irq` instead of `spin_lock` and `spin_unlock` .

Or for example `write_seqlock_irqsave` and `write_sequnlock_irqrestore` functions which are the same but used `spin_lock_irqsave` and `spin_unlock_irqsave` macro to use in IRQ) handlers.

That's all.

Conclusion

This is the end of the sixth part of the synchronization primitives chapter in the Linux kernel. In this part we met with new synchronization primitive which is called - `sequential lock` . From the theoretical side, this synchronization primitive very similar on a readers-writer lock synchronization primitive, but allows to avoid `writer-starving` issue.

If you have questions or suggestions, feel free to ping me in twitter 0xAX, drop me email or just create issue.

Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-insides.

Links

- synchronization primitives)
- readers-writer lock
- spinlock
- critical section
- lock validator
- debugging
- API
- x86_64
- Timers and time management in the Linux kernel
- interrupt handlers
- softirq
- IRQ)
- Previous part

Linux 内核内存管理

本章描述 Linux 内核中的内存管理。在本章中你会看到一系列描述 Linux 内核内存管理框架的不同部分的帖子。

- 内存块 - 描述早期的 `memblock` 分配器。
- 固定映射地址和 ioremap - 描述固定映射的地址和早期的 `ioremap` 。
- kmemcheck - 第三部分描述 `kmemcheck` 工具。

内核内存管理. 第一部分.

简介

内存管理是操作系统内核中最复杂的部分之一(我认为没有之一)。在讲解内核进入点之前的准备工作时，我们在调用 `start_kernel` 函数前停止了讲解。`start_kernel` 函数在内核启动第一个 `init` 进程前初始化了所有的内核特性(包括那些依赖于架构的特性)。你也许还记得在引导时建立了初期页表、识别页表和固定映射页表，但是复杂的内存管理部分还没有开始工作。当 `start_kernel` 函数被调用时，我们会看到从初期内存管理到更复杂的内存管理数据结构和技术的转变。为了更好地理解内核的初始化过程，我们需要对这些技术有更清晰的理解。本章节是内存管理框架和 API 的不同部分的概述，从 `memblock` 开始。

内存块

内存块是在引导初期，泛用内核内存分配器还没有开始工作时对内存区域进行管理的方法之一。以前它被称为 `逻辑内存块`，但是内核接纳了 Yinghai Lu 提供的补丁后改名为 `memblock`。`x86_64` 架构上的内核会使用这个方法。我们已经在讲解内核进入点之前的准备工作时遇到过了它。现在是时候对它更加熟悉了。我们会看到它是被怎样实现的。

我们首先会学习 `memblock` 的数据结构。以下所有的数据结构都在 include/linux/memblock.h 头文件中定义。

第一个结构体的名字就叫做 `memblock`。它的定义如下：

```
struct memblock {
         bool bottom_up;
         phys_addr_t current_limit;
         struct memblock_type memory;   --> array of memblock_region
         struct memblock_type reserved; --> array of memblock_region
#ifdef CONFIG_HAVE_MEMBLOCK_PHYS_MAP
         struct memblock_type physmem;
#endif
};
```

这个结构体包含五个域。第一个 `bottom_up` 域置为 `true` 时允许内存以自底向上模式进行分配。下一个域是 `current_limit`。 这个域描述了内存块的尺寸限制。接下来的三个域描述了内存块的类型。内存块的类型可以是：被保留，内存和物理内存(如果 `CONFIG_HAVE_MEMBLOCK_PHYS_MAP` 编译配置选项被开启)。接下来我们来看看下一个数据结构-`memblock_type`。让我们来看看它的定义：

```
struct memblock_type {
    unsigned long cnt;
    unsigned long max;
    phys_addr_t total_size;
    struct memblock_region *regions;
};
```

这个结构体提供了关于内存类型的信息。它包含了描述当前内存块中内存区域的数量、所有内存区域的大小、内存区域的已分配数组的尺寸和指向 `memblock_region` 结构体数据的指针的域。`memblock_region` 结构体描述了一个内存区域，定义如下：

```
struct memblock_region {
        phys_addr_t base;
        phys_addr_t size;
        unsigned long flags;
#ifdef CONFIG_HAVE_MEMBLOCK_NODE_MAP
        int nid;
#endif
};
```

`memblock_region` 提供了内存区域的基址和大小，`flags` 域可以是：

```
#define MEMBLOCK_ALLOC_ANYWHERE    (~(phys_addr_t)0)
#define MEMBLOCK_ALLOC_ACCESSIBLE    0
#define MEMBLOCK_HOTPLUG    0x1
```

同时，如果 `CONFIG_HAVE_MEMBLOCK_NODE_MAP` 编译配置选项被开启，`memblock_region` 结构体也提供了整数域 - numa 节点选择器。

我们将以上部分想象为如下示意图：

```
+---------------------------+   +---------------------------+
|         memblock          |   |                           |
|  _______________________  |   |                           |
| |        memory         | |   |       Array of the       |
| |      memblock_type    |-|-->|      membock_region       |
| |_______________________| |   |                           |
|                           |   +---------------------------+
|  _______________________  |   +---------------------------+
| |       reserved        | |   |                           |
| |      memblock_type    |-|-->|       Array of the       |
| |_______________________| |   |      memblock_region      |
|                           |   |                           |
+---------------------------+   +---------------------------+
```

这三个结构体： `memblock` ， `memblock_type` 和 `memblock_region` 是 `Memblock` 的主要组成部分。现在我们可以进一步了解 `Memblock` 和 它的初始化过程了。

内存块初始化

所有 `memblock` 的 API 都在 include/linux/memblock.h 头文件中描述, 所有函数的实现都在 mm/memblock.c 源码中。首先我们来看一下源码的开头部分和 `memblock` 结构体的初始化吧。

```
struct memblock memblock __initdata_memblock = {
    .memory.regions        = memblock_memory_init_regions,
    .memory.cnt            = 1,
    .memory.max            = INIT_MEMBLOCK_REGIONS,

    .reserved.regions    = memblock_reserved_init_regions,
    .reserved.cnt        = 1,
    .reserved.max        = INIT_MEMBLOCK_REGIONS,

#ifdef CONFIG_HAVE_MEMBLOCK_PHYS_MAP
    .physmem.regions    = memblock_physmem_init_regions,
    .physmem.cnt        = 1,
    .physmem.max        = INIT_PHYSMEM_REGIONS,
#endif
    .bottom_up            = false,
    .current_limit        = MEMBLOCK_ALLOC_ANYWHERE,
};
```

在这里我们可以看到 `memblock` 结构体的同名变量的初始化。首先请注意 `__initdata_memblock` 。这个宏的定义就像这样：

```
#ifdef CONFIG_ARCH_DISCARD_MEMBLOCK
    #define __init_memblock __meminit
    #define __initdata_memblock __meminitdata
#else
    #define __init_memblock
    #define __initdata_memblock
#endif
```

你会发现这个宏依赖于 `CONFIG_ARCH_DISCARD_MEMBLOCK` 。如果这个编译配置选项开启，内存块的代码会被放置在 `.init` 段，这样它就会在内核引导完毕后被释放掉。

接下来我们可以看看 `memblock_type memory` ， `memblock_type reserved` 和 `memblock_type physmem` 域的初始化。在这里我们只对 `memblock_type.regions` 的初始化过程感兴趣，请注意每一个 `memblock_type` 域都是 `memblock_region` 的数组初始化的：

```
static struct memblock_region memblock_memory_init_regions[INIT_MEMBLOCK_REGIONS] __in
itdata_memblock;
static struct memblock_region memblock_reserved_init_regions[INIT_MEMBLOCK_REGIONS] __
initdata_memblock;
#ifdef CONFIG_HAVE_MEMBLOCK_PHYS_MAP
static struct memblock_region memblock_physmem_init_regions[INIT_PHYSMEM_REGIONS] __in
itdata_memblock;
#endif
```

每个数组包含了 128 个内存区域。我们可以在 `INIT_MEMBLOCK_REGIONS` 宏定义中看到它：

```
#define INIT_MEMBLOCK_REGIONS   128
```

请注意所有的数组定义中也用到了在 `memblock` 中使用过的 `__initdata_memblock` 宏(如果忘掉了就翻到上面重温一下)。

最后两个域描述了 `bottom_up` 分配是否被开启以及当前内存块的限制：

```
#define MEMBLOCK_ALLOC_ANYWHERE (~(phys_addr_t)0)
```

这个限制是 `0xffffffffffffffff`.

On this step the initialization of the `memblock` structure has been finished and we can look on the Memblock API. 到此为止 `memblock` 结构体的初始化就结束了，我们可以开始看内存块相关 API 了。

内存块应用程序接口

我们已经结束了 `memblock` 结构体的初始化讲解，现在我们要开始看内存块 API 和它的实现了。就像我上面说过的，所有 `memblock` 的实现都在 mm/memblock.c 中。为了理解 `memblock` 是怎样被实现和工作的，让我们先看看它的用法。内核中有很多地方用到了内存块。举个例子，我们来看看 arch/x86/kernel/e820.c 中的 `memblock_x86_fill` 函数。这个函数使用了 e820 提供的内存映射并使用 `memblock_add` 函数在 `memblock` 中添加了内核保留的内存区域。既然我们首先遇到了 `memblock_add` 函数，让我们从它开始讲解吧。

这个函数获取了物理基址和内存区域的大小并把它们加到了 `memblock` 中。`memblock_add` 函数本身没有做任何特殊的事情，它只是调用了

```
memblock_add_range(&memblock.memory, base, size, MAX_NUMNODES, 0);
```

函数。我们将内存块类型 - `memory`，内存基址和内存区域大小，节点的最大数目和标志传进去。如果 `CONFIG_NODES_SHIFT` 没有被设置，最大节点数目就是 1，否则是 `1 << CONFIG_NODES_SHIFT`。`memblock_add_range` 函数将新的内存区域加到了内存块中，它首先检查传入内存区域的大小，如果是 0 就直接返回。然后，这个函数会用 `memblock_type` 来检查 `memblock` 中的内存区域是否存在。如果不存在，我们就简单地用给定的值填充一个新的 `memory_region` 然后返回(我们已经在对内核内存管理框架的初览中看到了它的实现)。如果 `memblock_type` 不为空，我们就会使用提供的 `memblock_type` 将新的内存区域加到 `memblock` 中。

首先，我们获取了内存区域的结束点：

```
phys_addr_t end = base + memblock_cap_size(base, &size);
```

`memblock_cap_size` 调整了 `size` 使 `base + size` 不会溢出。它的实现非常简单：

```
static inline phys_addr_t memblock_cap_size(phys_addr_t base, phys_addr_t *size)
{
    return *size = min(*size, (phys_addr_t)ULLONG_MAX - base);
}
```

`memblock_cap_size` 返回了提供的值与 `ULLONG_MAX - base` 中的较小值作为新的尺寸。

之后，我们获得了新的内存区域的结束地址，`memblock_add_range` 会检查与已加入内存区域是否重叠以及能否合并。将新的内存区域插入 `memblock` 包含两步：

- 将新内存区域的不重叠部分作为单独的区域加入；
- 合并所有相接的区域。

我们会迭代所有的已存储内存区域来检查是否与新区域重叠：

```
    for (i = 0; i < type->cnt; i++) {
        struct memblock_region *rgn = &type->regions[i];
        phys_addr_t rbase = rgn->base;
        phys_addr_t rend = rbase + rgn->size;

        if (rbase >= end)
            break;
        if (rend <= base)
            continue;
        ...
        ...
        ...
    }
```

如果新的内存区域不与已有区域重叠，直接插入。否则我们会检查这个新内存区域是否合适并调用 `memblock_double_array` 函数：

```
while (type->cnt + nr_new > type->max)
    if (memblock_double_array(type, obase, size) < 0)
        return -ENOMEM;
    insert = true;
    goto repeat;
```

`memblock_double_array` 会将提供的区域数组长度加倍。然后我们会将 `insert` 置为 `true`，接着跳转到 `repeat` 标签。第二步，我们会从 `repeat` 标签开始，迭代同样的循环然后使用 `memblock_insert_region` 函数将当前内存区域插入内存块：

```
    if (base < end) {
        nr_new++;
        if (insert)
            memblock_insert_region(type, i, base, end - base,
                                   nid, flags);
    }
```

我们在第一步将 `insert` 置为 `true`，现在 `memblock_insert_region` 会检查这个标志。`memblock_insert_region` 的实现与我们将新区域插入空 `memblock_type` 的实现(看上面)几乎相同。这个函数会获取最后一个内存区域：

```
struct memblock_region *rgn = &type->regions[idx];
```

然后用 `memmove` 拷贝这部分内存：

```
memmove(rgn + 1, rgn, (type->cnt - idx) * sizeof(*rgn));
```

之后我们会填充 `memblock_region` 域，然后增长 `memblock_type` 的尺寸。在函数执行的结束，`memblock_add_range` 会调用 `memblock_merge_regions` 来在第二步合并相邻可合并的内存区域。

还有第二种情况，新的内存区域与已储存区域完全重叠。比如 `memblock` 中已经有了 `region1`：

```
0                    0x1000
+-----------------------+
|                       |
|                       |
|        region1        |
|                       |
|                       |
+-----------------------+
```

现在我们想在 `memblock` 中添加 `region2` ，它的基址和尺寸如下：

```
0x100                 0x2000
+-----------------------+
|                       |
|                       |
|        region2        |
|                       |
|                       |
+-----------------------+
```

在这种情况下，新内存区域的基址会被像下面这样设置：

```
base = min(rend, end);
```

所以在我们设置的这种场景中，它会被设置为 `0x1000` 。然后我们会在第二步中将这个区域插入：

```
if (base < end) {
    nr_new++;
    if (insert)
        memblock_insert_region(type, i, base, end - base, nid, flags);
}
```

在这种情况下我们会插入 `overlapping portion` (我们之插入地址高的部分，因为低地址部分已经被包含在重叠区域里了)，然后会使用 `memblock_merge_regions` 合并剩余部分区域。就像我上文中所说的那样，这个函数会合并相邻的可合并区域。它会从给定的 `memblock_type` 遍历所有的内存区域，取出两个相邻区域 - `type->regions[i]` 和 `type->regions[i + 1]` ，并检查他们是否拥有同样的标志，是否属于同一个节点，第一个区域的末尾地址是否与第二个区域的基地址相同。

```
while (i < type->cnt - 1) {
    struct memblock_region *this = &type->regions[i];
    struct memblock_region *next = &type->regions[i + 1];
    if (this->base + this->size != next->base ||
        memblock_get_region_node(this) !=
        memblock_get_region_node(next) ||
        this->flags != next->flags) {
        BUG_ON(this->base + this->size > next->base);
        i++;
        continue;
    }
```

如果上面所说的这些条件全部符合，我们就会更新第一个区域的长度，将第二个区域的长度加上去。

```
this->size += next->size;
```

我们在更新第一个区域的长度同时，会使用 `memmove` 将后面的所有区域向前移动一个下标。

```
memmove(next, next + 1, (type->cnt - (i + 2)) * sizeof(*next));
```

然后将 `memblock_type` 中内存区域的数量减一：

```
type->cnt--;
```

经过这些操作后我们就成功地将两个内存区域合并了：

```
0                                              0x2000
+------------------------------------------------+
|                                                |
|                                                |
|                    region1                     |
|                                                |
|                                                |
+------------------------------------------------+
```

这就是 `memblock_add_range` 函数的工作原理和执行过程。

同样还有一个 `memblock_reserve` 函数与 `memblock_add` 几乎完成同样的工作，只有一点不同： `memblock_reserve` 将 `memblock_type.reserved` 而不是 `memblock_type.memory` 储存到内存块中。

当然这不是全部的 API。内存块不仅提供了添加 `memory` 和 `reserved` 内存区域，还提供了：

- memblock_remove - 从内存块中移除内存区域；
- memblock_find_in_range - 寻找给定范围内的未使用区域；
- memblock_free - 释放内存块中的内存区域；
- for_each_mem_range - 迭代遍历内存块区域。

等等......

获取内存区域的相关信息

内存块还提供了获取 `memblock` 中已分配内存区域信息的 API。包括两部分：

- get_allocated_memblock_memory_regions_info - 获取有关内存区域的信息；
- get_allocated_memblock_reserved_regions_info - 获取有关保留区域的信息。

这些函数的实现都很简单。以 `get_allocated_memblock_reserved_regions_info` 为例：

```
phys_addr_t __init_memblock get_allocated_memblock_reserved_regions_info(
                    phys_addr_t *addr)
{
    if (memblock.reserved.regions == memblock_reserved_init_regions)
        return 0;

    *addr = __pa(memblock.reserved.regions);

    return PAGE_ALIGN(sizeof(struct memblock_region) *
              memblock.reserved.max);
}
```

这个函数首先会检查 `memblock` 是否包含保留内存区域。如果否，就直接返回 0 。否则函数将保留内存区域的物理地址写到传入的数组中，然后返回已分配数组的对齐后尺寸。注意函数使用 `PAGE_ALIGN` 这个宏实现对齐。实际上这个宏依赖于页的尺寸：

```
#define PAGE_ALIGN(addr) ALIGN(addr, PAGE_SIZE)
```

`get_allocated_memblock_memory_regions_info` 函数的实现是基本一样的。只有一处不同，`get_allocated_memblock_memory_regions_info` 使用 `memblock_type.memory` 而不是 `memblock_type.reserved` 。

内存块的相关除错技术

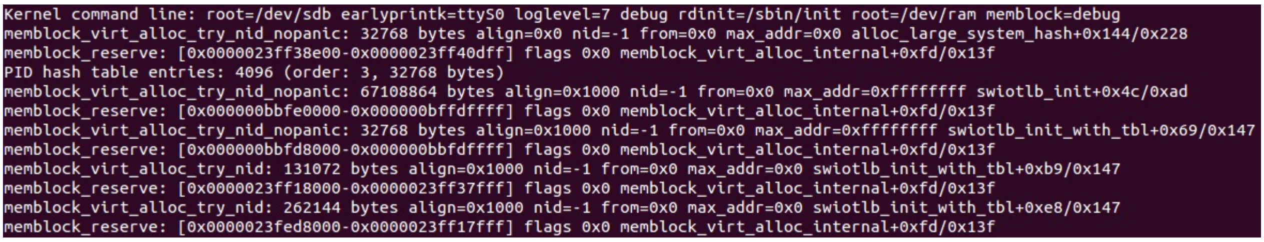
在内存块的实现中有许多对 `memblock_dbg` 的调用。如果在内核命令行中传入 `memblock=debug` 选项，这个函数就会被调用。实际上 `memblock_dbg` 是 `printk` 的一个拓展宏：

```
#define memblock_dbg(fmt, ...) \
         if (memblock_debug) printk(KERN_INFO pr_fmt(fmt), ##__VA_ARGS__)
```

比如你可以在 `memblock_reserve` 函数中看到对这个宏的调用：

```
memblock_dbg("memblock_reserve: [%#016llx-%#016llx] flags %#02lx %pF\n",
             (unsigned long long)base,
             (unsigned long long)base + size - 1,
             flags, (void *)_RET_IP_);
```

然后你将看到类似下图的画面：



内存块技术也支持 debugfs 。如果你不是在 `X86` 架构下运行内核，你可以访问：

- /sys/kernel/debug/memblock/memory
- /sys/kernel/debug/memblock/reserved
- /sys/kernel/debug/memblock/physmem

来获取 `memblock` 内容的核心转储信息。

结束语

讲解内核内存管理的第一部分到此结束，如果你有任何的问题或者建议，你可以直接发消息给我twitter，也可以给我发邮件或是直接创建一个 issue。

英文不是我的母语。如果你发现我的英文描述有任何问题，请提交一个**PR**到**linux-insides.**

相关连接：

- e820
- numa
- debugfs

- 对内核内存管理框架的初览

内核内存管理. 第二部分.

固定映射地址和输入输出重映射

固定映射地址是一组特殊的编译时确定的地址，它们与物理地址不一定具有减 `__START_KERNEL_map` 的线性映射关系。每一个固定映射的地址都会映射到一个内存页，内核会像指针一样使用它们，但是绝不会修改它们的地址。这是这种地址的主要特点。就像注释所说的那样，“在编译期就获得一个常量地址，只有在引导阶段才会被设定上物理地址。”你在本书的前面部分可以看到，我们已经设定了 `level2_fixmap_pgt` ：

```
NEXT_PAGE(level2_fixmap_pgt)
    .fill    506,8,0
    .quad    level1_fixmap_pgt - __START_KERNEL_map + _PAGE_TABLE
    .fill    5,8,0

NEXT_PAGE(level1_fixmap_pgt)
    .fill    512,8,0
```

就像我们看到的， `level2_fixmap_pgt` 紧挨着 `level2_kernel_pgt` 保存了内核的 code+data+bss 段。每一个固定映射的地址都由一个整数下标表示，这些整数下标在 arch/x86/include/asm/fixmap.h 的 `fixed_addresses` 枚举类型中定义。比如，它包含了 `VSYSCALL_PAGE` 的入口 - 如果合法的 vsyscall 页模拟机制被开启，或是启用了本地 apic 的 `FIX_APIC_BASE` 选项等等。在虚拟内存中，固定映射区域被放置在模块区域中：

```
         +-----------+-----------------+---------------+------------------+
         |           |                 |               |                  |
         |kernel text|      kernel     |               |    vsyscalls     |
         | mapping   |       text      |    Modules    |    fix-mapped    |
         |from phys 0|       data      |               |    addresses     |
         |           |                 |               |                  |
         +-----------+-----------------+---------------+------------------+
__START_KERNEL_map   __START_KERNEL    MODULES_VADDR            0xffffffffffffffff
```

基虚拟地址和固定映射区域的尺寸使用以下两个宏表示：

```
#define FIXADDR_SIZE    (__end_of_permanent_fixed_addresses << PAGE_SHIFT)
#define FIXADDR_START        (FIXADDR_TOP - FIXADDR_SIZE)
```

在这里 `__end_of_permanent_fixed_addresses` 是 `fixed_addresses` 枚举中的一个元素，如我上文所说：每一个固定映射地址都由一个定义在 `fixed_addresses` 中的整数下标表示。`PAGE_SHIFT` 决定了页的大小。比如，我们可以使用 `1 << PAGE_SHIFT` 来获取一页的大小。在我们的场景下需要获取固定映射区域的尺寸，而不仅仅是一页的大小，这就是我们使用 `__end_of_permanent_fixed_addresses` 来获取固定映射区域尺寸的原因。在我的系统中这个值可能略大于 `536` KB。在你的系统上这个值可能会不同，因为这个值取决于固定映射地址的数目，而这个数目又取决于内核的配置。

The second `FIXADDR_START` macro just substracts fix-mapped area size from the last address of the fix-mapped area to get its base virtual address. `FIXADDR_TOP` is a rounded up address from the base address of the vsyscall space: 第二个 `FIXADDR_START` 宏只是从固定映射区域的末地址减去了固定映射区域的尺寸，这样就可以获得它的基虚拟地址。`FIXADDR_TOP` 是一个从 vsyscall 空间的基址取整产生的地址：

```
#define FIXADDR_TOP     (round_up(VSYSCALL_ADDR + PAGE_SIZE, 1<<PMD_SHIFT) - PAGE_SIZE)
```

`fixed_addresses` 枚举量被 `fix_to_virt` 函数用做下标用于获取虚拟地址。这个函数的实现很简单：

```
static __always_inline unsigned long fix_to_virt(const unsigned int idx)
{
        BUILD_BUG_ON(idx >= __end_of_fixed_addresses);
        return __fix_to_virt(idx);
}
```

首先它调用 `BUILD_BUG_ON` 宏检查了给定的 `fixed_addresses` 枚举量不大于等于 `__end_of_fixed_addresses`，然后返回了 `__fix_to_virt` 宏的运算结果：

```
#define __fix_to_virt(x)        (FIXADDR_TOP - ((x) << PAGE_SHIFT))
```

在这里我们用 `PAGE_SHIFT` 左移了给定的固定映射地址下标，就像我上文所述它决定了页的地址，然后将 `FIXADDR_TOP` 减去这个值，`FIXADDR_TOP` 是固定映射区域的最高地址。以下是从虚拟地址获取对应固定映射地址的转换函数：

```
static inline unsigned long virt_to_fix(const unsigned long vaddr)
{
        BUG_ON(vaddr >= FIXADDR_TOP || vaddr < FIXADDR_START);
        return __virt_to_fix(vaddr);
}
```

`virt_to_fix` 以虚拟地址为参数，检查了这个地址是否位于 `FIXADDR_START` 和 `FIXADDR_TOP` 之间，然后调用 `__virt_to_fix` ，这个宏实现如下：

```
#define __virt_to_fix(x)        ((FIXADDR_TOP - ((x)&PAGE_MASK)) >> PAGE_SHIFT)
```

一个 PFN 是一块页大小物理内存的下标。一个物理地址的 PFN 可以简单地定义为 (page_phys_addr >> PAGE_SHIFT)；

`__virt_to_fix` 会清空给定地址的前 12 位，然后用固定映射区域的末地址(`FIXADDR_TOP`)减去它并右移 `PAGE_SHIFT` 即 12 位。让我们来解释它的工作原理。就像我已经写的那样，这个宏会使用 `x & PAGE_MASK` 来清空前 12 位。然后我们用 `FIXADDR_TOP` 减去它，就会得到 `FIXADDR_TOP` 的后 12 位。我们知道虚拟地址的前 12 位代表这个页的偏移量，当我们右移 `PAGE_SHIFT` 后就会得到 `Page frame number` ，即虚拟地址的所有位，包括最开始的 12 个偏移位。固定映射地址在内核中多处使用。 `IDT` 描述符保存在这里，英特尔可信赖执行技术 UUID 储存在固定映射区域，以 `FIX_TBOOT_BASE` 下标开始。另外， Xen 引导映射等也储存在这个区域。我们已经在内核初始化的第五部分看到了一部分关于固定映射地址的知识。接下来让我们看看什么是 `ioremap` ，看看它是怎样实现的，与固定映射地址又有什么关系呢？

输入输出重映射

内核提供了许多不同的内存管理原语。现在我们将要接触 `I/O 内存` 。每一个设备都通过读写它的寄存器来控制。比如，驱动可以通过向它的寄存器中写来打开或关闭设备，也可以通过读它的寄存器来获取设备状态。除了寄存器之外，许多设备都拥有一块可供驱动读写的缓冲区。如我们所知，现在有两种方法来访问设备的寄存器和数据缓冲区：

- 通过 I/O 端口；
- 将所有寄存器映射到内存地址空间；

第一种情况，设备的所有控制寄存器都具有一个输入输出端口号。该设备的驱动可以用 `in` 和 `out` 指令来从端口中读写。你可以通过访问 `/proc/ioports` 来获取设备当前的 I/O 端口号。

```
$ cat /proc/ioports
0000-0cf7 : PCI Bus 0000:00
  0000-001f : dma1
  0020-0021 : pic1
  0040-0043 : timer0
  0050-0053 : timer1
  0060-0060 : keyboard
  0064-0064 : keyboard
  0070-0077 : rtc0
  0080-008f : dma page reg
  00a0-00a1 : pic2
  00c0-00df : dma2
  00f0-00ff : fpu
    00f0-00f0 : PNP0C04:00
  03c0-03df : vesafb
  03f8-03ff : serial
  04d0-04d1 : pnp 00:06
  0800-087f : pnp 00:01
  0a00-0a0f : pnp 00:04
  0a20-0a2f : pnp 00:04
  0a30-0a3f : pnp 00:04
0cf8-0cff : PCI conf1
0d00-ffff : PCI Bus 0000:00
...
...
...
```

`/proc/ioports` 提供了驱动使用 I/O 端口的内存区域地址。所有的这些内存区域，比如 `0000-0cf7` ，都是使用 include/linux/ioport.h 头文件中的 `request_region` 来声明的。实际上 `request_region` 是一个宏，它的定义如下：

```
#define request_region(start,n,name)   __request_region(&ioport_resource, (start), (n)
, (name), 0)
```

正如我们所看见的，它有三个参数：

- `start` - 区域的起点;
- `n` - 区域的长度;
- `name` - 区域需求者的名字。

`request_region` 分配 I/O 端口区域。通常在 `request_region` 之前会调用 `check_region` 来检查传入的地址区间是否可用，然后 `release_region` 会释放这个内存区域。 `request_region` 返回指向 `resource` 结构体的指针。 `resource` 结构体是对系统资源的树状子集的抽象。我们已经在内核初始化的第五部分见到过它了，它的定义是这样的：

```
struct resource {
        resource_size_t start;
        resource_size_t end;
        const char *name;
        unsigned long flags;
        struct resource *parent, *sibling, *child;
};
```

它包含起止地址、名字等等。每一个 `resource` 结构体包含一个指向 `parent` 、 `slibling` 和 `child` 资源的指针。它有父节点和子节点，这就意味着每一个资源的子集都有一个根节点。比如，对 I/O 端口来说有一个 `ioport_resource` 结构体：

```
struct resource ioport_resource = {
         .name   = "PCI IO",
         .start  = 0,
         .end    = IO_SPACE_LIMIT,
        .flags  = IORESOURCE_IO,
};
EXPORT_SYMBOL(ioport_resource);
```

或者对 `iomem` 来说，有一个 `iomem_resource` 结构体：

```
struct resource iomem_resource = {
        .name   = "PCI mem",
        .start  = 0,
        .end    = -1,
        .flags  = IORESOURCE_MEM,
};
```

就像我所写的， `request_region` 用于注册 I/O 端口区域，这个宏用于内核中的许多地方。比如让我们来看看 drivers/char/rtc.c。这个源文件提供了内核中的实时时钟接口。与其他内核模块一样， `rtc` 模块包含一个 `module_init` 定义：

```
module_init(rtc_init);
```

在这里 `rtc_init` 是 `rtc` 模块的初始化函数。这个函数也定义在 `rtc.c` 文件中。在 `rtc_init` 函数中我们可以看到许多对 `rtc_request_region` 函数的调用，实际上这是 `request_region` 的包装：

```
r = rtc_request_region(RTC_IO_EXTENT);
```

`rtc_request_region` 中调用了:

```
r = request_region(RTC_PORT(0), size, "rtc");
```

在这里 `RTC_TO_EXTENT` 是一个内存区域的尺寸，在这里是 `0x8`， `"rtc"` 是区域的名字， `RTC_PORT` 是：

```
#define RTC_PORT(x)     (0x70 + (x))
```

所以使用 `request_region(RTC_PORT(0), size, "rtc")` 我们注册了一个内存区域， 以 `0x70` 开始，大小为 `0x8` 。 让我们看看 `/proc/ioports`：

```
~$ sudo cat /proc/ioports | grep rtc
0070-0077 : rtc0
```

看，我们可以获取了它的信息。这就是端口。第二种途径是使用 I/O 内存。就像我上面写的，这是将设备的控制寄存器和内存映射到内存地址空间中。I/O 内存是一组由设备通过总线提供给 CPU 的相邻的地址。所有的 I/O 映射地址都不能由内核直接访问。有一个 `ioremap` 函数用来将总线上的物理地址转化为内核的虚拟地址，或者说， `ioremap` 映射了 I/O 物理地址来让他们能够在内核中使用。这个函数有两个参数：

- 内存区域的开始；
- 内存区域的结束；

I/O 内存映射 API 提供了用来检查、请求与释放内存区域的函数，就像 I/O 端口 API 一样。这里有三个函数：

- `request_mem_region`
- `release_mem_region`
- `check_mem_region`

```
~$ sudo cat /proc/iomem
...
...
...
be826000-be82cfff : ACPI Non-volatile Storage
be82d000-bf744fff : System RAM
bf745000-bfff4fff : reserved
bfff5000-dc041fff : System RAM
dc042000-dc0d2fff : reserved
dc0d3000-dc138fff : System RAM
dc139000-dc27dfff : ACPI Non-volatile Storage
dc27e000-deffefff : reserved
defff000-deffffff : System RAM
df000000-dfffffff : RAM buffer
e0000000-feafffff : PCI Bus 0000:00
  e0000000-efffffff : PCI Bus 0000:01
    e0000000-efffffff : 0000:01:00.0
  f7c00000-f7cfffff : PCI Bus 0000:06
    f7c00000-f7c0ffff : 0000:06:00.0
    f7c10000-f7c101ff : 0000:06:00.0
      f7c10000-f7c101ff : ahci
  f7d00000-f7dfffff : PCI Bus 0000:03
    f7d00000-f7d3ffff : 0000:03:00.0
      f7d00000-f7d3ffff : alx
...
...
...
```

这些地址中的一部分源于对 `e820_reserve_resources` 函数的调用。我们可以在 arch/x86/kernel/setup.c 中找到对这个函数的调用，这个函数本身定义在 arch/x86/kernel/e820.c 中。这个函数遍历了 e820 的映射然后将内存区域插入了根 `iomen` 结构体中。所有具有以下类型的 `e820` 内存区域都会被插入到 `iomem` 结构体中：

```
static inline const char *e820_type_to_string(int e820_type)
{
    switch (e820_type) {
    case E820_RESERVED_KERN:
    case E820_RAM:    return "System RAM";
    case E820_ACPI:    return "ACPI Tables";
    case E820_NVS:    return "ACPI Non-volatile Storage";
    case E820_UNUSABLE:    return "Unusable memory";
    default:    return "reserved";
    }
}
```

我们可以在 `/proc/iomem` 中看到它们。

现在让我们尝试着理解 `ioremap` 是如何工作的。我们已经了解了一部分 `ioremap` 的知识，我们在内核初始化的第五部分见过它。如果你读了那个章节，你就会记得 arch/x86/mm/ioremap.c 文件中对 `early_ioremap_init` 函数的调用。对 `ioremap` 的初始化分为两个部分：有一部分在我们正常使用 `ioremap` 之前，但是要首先进行 `vmalloc` 的初始化并调用 `paging_init` 才能进行正常的 `ioremap` 调用。我们现在还不了解 `vmalloc` 的知识，先看看第一部分的初始化。首先 `early_ioremap_init` 会检查固定映射是否与页中部目录对齐：

```
BUILD_BUG_ON((fix_to_virt(0) + PAGE_SIZE) & ((1 << PMD_SHIFT) - 1));
```

更多关于 `BUILD_BUG_ON` 的内容你可以在内核初始化的第一部分看到。如果给定的表达式为真，`BUILD_BUG_ON` 宏就会抛出一个编译时错误。在检查后的下一步，我们可以看到对 `early_ioremap_setup` 函数的调用，这个函数定义在 mm/early_ioremap.c 文件中。这个函数代表了对 `ioremap` 的大体初始化。`early_ioremap_setup` 函数用初期固定映射的地址填充了 `slot_virt` 数组。所有初期固定映射地址在内存中都在 `__end_of_permanent_fixed_addresses` 后面，它们从 `FIX_BITMAP_BEGIN` 开始，到 `FIX_BITMAP_END` 结束。实际上初期 `ioremap` 会使用 `512` 个临时引导时映射：

```
#define NR_FIX_BTMAPS		64
#define FIX_BTMAPS_SLOTS	8
#define TOTAL_FIX_BTMAPS	(NR_FIX_BTMAPS * FIX_BTMAPS_SLOTS)
```

`early_ioremap_setup` 如下：

```
void __init early_ioremap_setup(void)
{
        int i;

        for (i = 0; i < FIX_BTMAPS_SLOTS; i++)
                if (WARN_ON(prev_map[i]))
                        break;

        for (i = 0; i < FIX_BTMAPS_SLOTS; i++)
                slot_virt[i] = __fix_to_virt(FIX_BTMAP_BEGIN - NR_FIX_BTMAPS*i);
}
```

`slot_virt` 和其他数组定义在同一个源文件中：

```
static void __iomem *prev_map[FIX_BTMAPS_SLOTS] __initdata;
static unsigned long prev_size[FIX_BTMAPS_SLOTS] __initdata;
static unsigned long slot_virt[FIX_BTMAPS_SLOTS] __initdata;
```

`slot_virt` 包含了固定映射区域的虚拟地址，`prev_map` 数组包含了初期 `ioremap` 区域的地址。注意我在上文中提到的：`实际上初期 ioremap 会使用 512 个临时引导时映射`，同时你可以看到所有的数组都使用 `__initdata` 定义，这意味着这些内存都会在内核初始化结束后释放掉。在 `early_ioremap_setup` 结束后，我们获得了页中部目录，以 `early_ioremap_pmd` 函数开始的早期 `ioremap`，`early_ioremap_pmd` 函数只能获得内存全局目录以及为给定地址计算页中部目录：

```
static inline pmd_t * __init early_ioremap_pmd(unsigned long addr)
{
    pgd_t *base = __va(read_cr3());
    pgd_t *pgd = &base[pgd_index(addr)];
    pud_t *pud = pud_offset(pgd, addr);
    pmd_t *pmd = pmd_offset(pud, addr);
    return pmd;
}
```

之后我们用 0 填充 `bm_pte` (早期 `ioremap` 页表入口)，然后调用 `pmd_populate_kernel` 函数：

```
pmd = early_ioremap_pmd(fix_to_virt(FIX_BTMAP_BEGIN));
memset(bm_pte, 0, sizeof(bm_pte));
pmd_populate_kernel(&init_mm, pmd, bm_pte);
```

`pmd_populate_kernel` 函数有三个参数:

- `init_mm` - `init` 进程的内存描述符 (你可以在前文中看到)；
- `pmd` - `ioremap` 固定映射开始处的页中部目录；
- `bm_pte` - 初期 `ioremap` 页表入口数组定义为：

```
static pte_t bm_pte[PAGE_SIZE/sizeof(pte_t)] __page_aligned_bss;
```

`pmd_popularte_kernel` 函数定义在 arch/x86/include/asm/pgalloc.h 中。它会用给定的页表入口(`bm_pte`)生成给定页中部目录(`pmd`):

```
static inline void pmd_populate_kernel(struct mm_struct *mm,
                                       pmd_t *pmd, pte_t *pte)
{
        paravirt_alloc_pte(mm, __pa(pte) >> PAGE_SHIFT);
        set_pmd(pmd, __pmd(__pa(pte) | _PAGE_TABLE));
}
```

`set_pmd` 声明如下：

```
#define set_pmd(pmdp, pmd)              native_set_pmd(pmdp, pmd)
```

`native_set_pmd` 声明如下：

```
static inline void native_set_pmd(pmd_t *pmdp, pmd_t pmd)
{
        *pmdp = pmd;
}
```

到这里 初期 `ioremap` 就可以使用了。在 `early_ioremap_init` 函数中有许多检查，但是都不重要，总之 `ioremap` 的初始化结束了。

初期输入输出重映射的使用

初期 `ioremap` 初始化完成后，我们就能使用它了。它提供了两个函数：

- early_ioremap
- early_iounmap

用于从 IO 物理地址 映射/解除映射 到虚拟地址。这俩函数都依赖于 `CONFIG_MMU` 编译配置选项。内存管理单元是内存管理的一种特殊块。这种块的主要用途是将物理地址转换为虚拟地址。技术上看内存管理单元可以从 `cr3` 控制寄存器中获取高等级页表地址(`pgd`)。如果 `CONFIG_MMU` 选项被设为 `n` ，`early_ioremap` 就会直接返回物理地址，而 `early_iounmap` 就会什么都不做。另一方面，如果设为 `y` ，`early_ioremap` 就会调用 `__early_ioremap` ，它有三个参数：

- `phys_addr` - 要映射到虚拟地址上的 I/O 内存区域的基物理地址；
- `size` - I/O 内存区域的尺寸；
- `prot` - 页表入口位。

在 `__early_ioremap` 中我们首先遍历了所有初期 `ioremap` 固定映射槽并检查 `prev_map` 数组中第一个空闲元素，然后将这个值存在了 `slot` 变量中，另外设置了尺寸：

```
slot = -1;
for (i = 0; i < FIX_BTMAPS_SLOTS; i++) {
    if (!prev_map[i]) {
        slot = i;
        break;
    }
}
...
...
...
prev_size[slot] = size;
last_addr = phys_addr + size - 1;
```

在下一步中我们会看到以下代码：

```
offset = phys_addr & ~PAGE_MASK;
phys_addr &= PAGE_MASK;
size = PAGE_ALIGN(last_addr + 1) - phys_addr;
```

在这里我们使用了 `PAGE_MASK` 用于清空除前 12 位之外的整个 `phys_addr` 。 `PAGE_MASK` 宏定义如下：

```
#define PAGE_MASK       (~(PAGE_SIZE-1))
```

我们知道页的尺寸是 4096 个字节或用二进制表示为 `1000000000000` 。 `PAGE_SIZE - 1` 就会是 `111111111111` ，但是使用 `~` 运算后我们就会得到 `000000000000` ，然后使用 `~PAGE_MASK` 又会返回 `111111111111` 。在第二行我们做了同样的事情但是只是清空了前 12 个位，然后在第三行获取了这个区域的页对齐尺寸。我们获得了对齐区域，接下来就需要获取新的 `ioremap` 区域所占用的页的数量然后计算固定映射下标：

```
nrpages = size >> PAGE_SHIFT;
idx = FIX_BTMAP_BEGIN - NR_FIX_BTMAPS*slot;
```

现在我们用给定的物理地址填充了固定映射区域。循环中的每一次迭代，我们都调用一次 arch/x86/mm/ioremap.c 中的 `__early_set_fixmap` 函数，为给定的物理地址加上页的大小 `4096` ，然后更新下标和页的数量：

```
while (nrpages > 0) {
    __early_set_fixmap(idx, phys_addr, prot);
    phys_addr += PAGE_SIZE;
    --idx;
    --nrpages;
}
```

`__early_set_fixmap` 函数为给定的物理地址获取了页表入口(保存在 `bm_pte` 中，见上文)：

```
pte = early_ioremap_pte(addr);
```

在 `early_ioremap_pte` 的下一步中我们用 `pgprot_val` 宏检查了给定的页标志，依赖这个标志选择调用 `set_pte` 还是 `pte_clear` ：

```
if (pgprot_val(flags))
        set_pte(pte, pfn_pte(phys >> PAGE_SHIFT, flags));
    else
        pte_clear(&init_mm, addr, pte);
```

As you can see above, we passed `FIXMAP_PAGE_IO` as flags to the `__early_ioremap`. `FIXMPA_PAGE_IO` expands to the: 就像你看到的，我们将 `FIXMAP_PAGE_IO` 作为标志传入了 `__early_ioremap` 。`FIXMPA_PAGE_IO` 从以下

```
(__PAGE_KERNEL_EXEC | _PAGE_NX)
```

标志拓展而来， 所以我们调用 `set_pte` 来设置页表入口，就像 `set_pmd` 一样，只不过用于 `PTE` (见上文)。我们在循环中设定了所有 `PTE` ，我们可以看到 `__flush_tlb_one` 的函数调用：

```
__flush_tlb_one(addr);
```

这个函数定义在 arch/x86/include/asm/tlbflush.h中，并通过判断 `cpu_has_invlpg` 的值来决定调用 `__flush_tlb_single` 还是 `__flush_tlb` ：

```
static inline void __flush_tlb_one(unsigned long addr)
{
        if (cpu_has_invlpg)
                __flush_tlb_single(addr);
        else
                __flush_tlb();
}
```

`__flush_tlb_one` 函数使 TLB 中的给定地址失效。就像你看到的我们更新了页结构，但是 `TLB` 还没有改变，这就是我们需要手动做这件事情的原因。有两种方法做这件事。第一种是更新 `cr3` 寄存器， `__flush_tlb` 函数就是这么做的：

```
native_write_cr3(native_read_cr3());
```

第二种方法是使用 `invlpg` 命令来使 `TLB` 入口失效。让我们看看 `__flush_tlb_one` 的实现。就像我们所看到的，它首先检查了 `cpu_has_invlpg` ，定义如下：

```
#if defined(CONFIG_X86_INVLPG) || defined(CONFIG_X86_64)
# define cpu_has_invlpg         1
#else
# define cpu_has_invlpg         (boot_cpu_data.x86 > 3)
#endif
```

如果 CPU 支持 `invlpg` 指令，我们就调用 `__flush_tlb_single` 宏，它拓展自 `__native_flush_tlb_single` ：

```
static inline void __native_flush_tlb_single(unsigned long addr)
{
        asm volatile("invlpg (%0)" ::"r" (addr) : "memory");
}
```

`__flush_tlb` 的调用知识更新了 `cr3` 寄存器。在这步结束之后 `__early_set_fixmap` 函数就执行完了，我们又可以回到 `__early_ioremap` 的实现了。因为我们为给定的地址设定了固定映射区域，我们需要将 I/O 重映射的区域的基虚拟地址用 `slot` 下标保存在 `prev_map` 数组中。

```
prev_map[slot] = (void __iomem *)(offset + slot_virt[slot]);
```

然后返回它。

第二个函数是 `early_iounmap` ，它会解除对一个 I/O 内存区域的映射。这个函数有两个参数：基地址和 I/O 区域的大小，这看起来与 `early_ioremap` 很像。它同样遍历了固定映射槽并寻找给定地址的槽。这样它就获得了这个固定映射槽的下标，然后通过判断 `after_paging_init` 的值决定是调用 `__late_clear_fixmap` 还是 `__early_set_fixmap` 。当这个值是 0 时会调用 `__early_set_fixmap` 。最终它会将 I/O 内存区域设为 `NULL` ：

```
prev_map[slot] = NULL;
```

这就是关于 `fixmap` 和 `ioremap` 的全部内容。当然这部分不可能包含所有 `ioremap` 的特性，仅仅是讲解了初期 `ioremap` ，常规的 `ioremap` 没有讲。这主要是因为在讲解它之前需要了解更多内容才行。

就是这样！

结束语

讲解内核内存管理的第一部分到此结束，如果你有任何的问题或者建议，你可以直接发消息给我twitter，也可以给我发邮件或是直接创建一个 issue。

英文不是我的母语。如果你发现我的英文描述有任何问题，请提交一个**PR**到**linux-insides.**

相关连接：

- apic
- vsyscall
- Intel Trusted Execution Technology
- Xen
- Real Time Clock
- e820
- Memory management unit
- TLB
- Paging
- 内核内存管理第一部分

Linux内核内存管理 第三节

内核中 kmemcheck 介绍

Linux内存管理章节描述了Linux内核中内存管理；本小节是第三部分。 在本章第二节中我们遇到了两个与内存管理相关的概念：

- `固定映射地址`；
- `输入输出重映射`.

固定映射地址代表虚拟内存中的一类特殊区域， 这类地址的物理映射地址是在编译期间计算出来的。输入输出重映射表示把输入/输出相关的内存映射到虚拟内存。

例如，查看 `/proc/iomem` 命令：

```
$ sudo cat /proc/iomem

00000000-00000fff : reserved
00001000-0009d7ff : System RAM
0009d800-0009ffff : reserved
000a0000-000bffff : PCI Bus 0000:00
000c0000-000cffff : Video ROM
000d0000-000d3fff : PCI Bus 0000:00
000d4000-000d7fff : PCI Bus 0000:00
000d8000-000dbfff : PCI Bus 0000:00
000dc000-000dffff : PCI Bus 0000:00
000e0000-000fffff : reserved
...
...
...
```

`iomem` 命令的输出显示了系统中每个物理设备所映射的内存区域。第一列为物理设备分配的内存区域，第二列为对应的各种不同类型的物理设备。再例如：

```
$ sudo cat /proc/ioports

0000-0cf7 : PCI Bus 0000:00
  0000-001f : dma1
  0020-0021 : pic1
  0040-0043 : timer0
  0050-0053 : timer1
  0060-0060 : keyboard
  0064-0064 : keyboard
  0070-0077 : rtc0
  0080-008f : dma page reg
  00a0-00a1 : pic2
  00c0-00df : dma2
  00f0-00ff : fpu
    00f0-00f0 : PNP0C04:00
  03c0-03df : vga+
  03f8-03ff : serial
  04d0-04d1 : pnp 00:06
  0800-087f : pnp 00:01
  0a00-0a0f : pnp 00:04
  0a20-0a2f : pnp 00:04
  0a30-0a3f : pnp 00:04
...
...
...
```

`ioports` 的输出列出了系统中物理设备所注册的各种类型的I/O端口。内核不能直接访问设备的输入/输出地址。在内核能够使用这些内存之前，必须将这些地址映射到虚拟地址空间，这就是 `io remap` 机制的主要目的。在前面第二节中只介绍了早期的 `io remap` 。很快我们就要来看一看常规的 `io remap` 实现机制。但在此之前，我们需要学习一些其他的知识，例如不同类型的内存分配器等，不然的话我们很难理解该机制。

在进入Linux内核常规期的内存管理之前，我们要看一些特殊的内存机制，例如调试，检查内存泄漏，内存控制等等。学习这些内容有助于我们理解Linux内核的内存管理。

从本节的标题中，你可能已经看出来，我们会从kmemcheck开始了解内存机制。和前面的章节一样，我们首先从理论上学习什么是 `kmemcheck` ，然后再来看Linux内核中是怎么实现这一机制的。

让我们开始吧。Linux内核中的 `kmemcheck` 到底是什么呢？从该机制的名称上你可能已经猜到， `kmemcheck` 是检查内存的。你猜的很对。 `kmemcheck` 的主要目的就是用来检查是否有内核代码访问 `未初始化的内存` 。让我们看一个简单的 C 程序：

```
#include <stdlib.h>
#include <stdio.h>

struct A {
        int a;
};

int main(int argc, char **argv) {
        struct A *a = malloc(sizeof(struct A));
        printf("a->a = %d\n", a->a);
        return 0;
}
```

在上面的程序中我们给结构体 `A` 分配了内存，然后我们尝试打印它的成员 `a`。如果我们不使用其他选项来编译该程序：

```
gcc test.c -o test
```

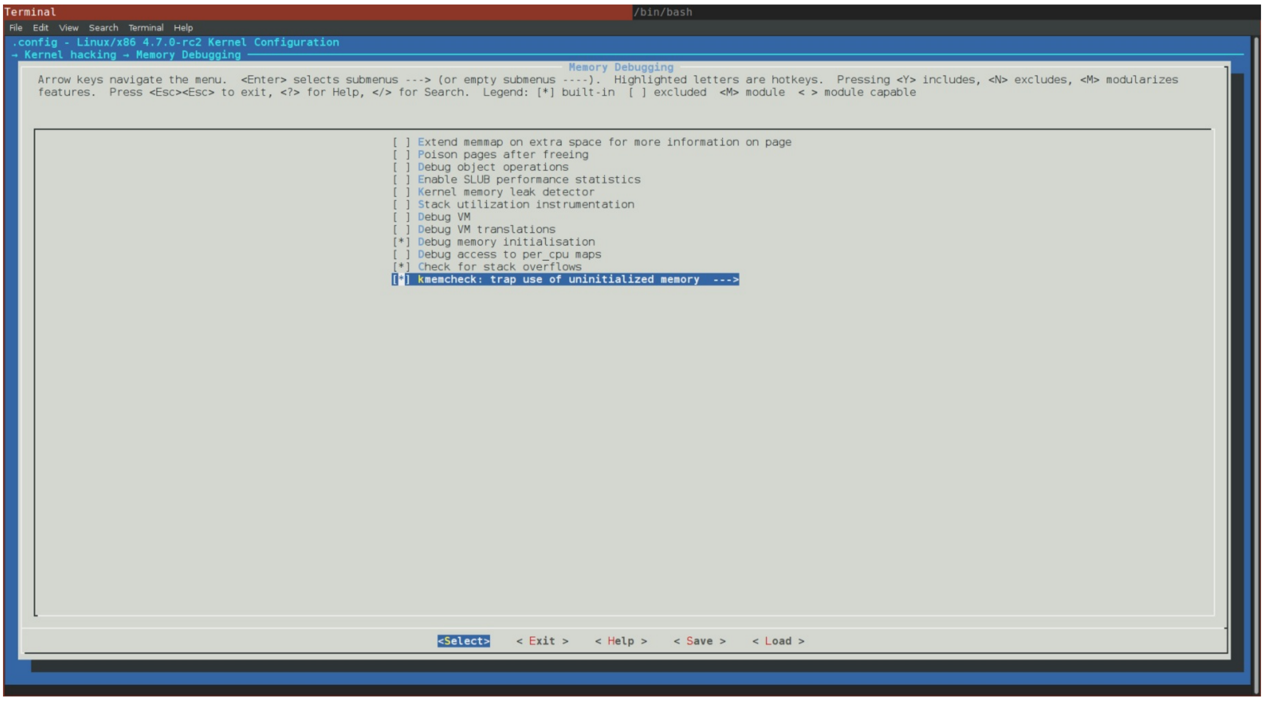
编译器不会显示成员 `a` 未初始化的提示信息。但是如果使用工具valgrind来运行该程序，我们会看到如下输出：

```
~$   valgrind --leak-check=yes ./test
==28469== Memcheck, a memory error detector
==28469== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==28469== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==28469== Command: ./test
==28469==
==28469== Conditional jump or move depends on uninitialised value(s)
==28469==    at 0x4E820EA: vfprintf (in /usr/lib64/libc-2.22.so)
==28469==    by 0x4E88D48: printf (in /usr/lib64/libc-2.22.so)
==28469==    by 0x4005B9: main (in /home/alex/test)
==28469==
==28469== Use of uninitialised value of size 8
==28469==    at 0x4E7E0BB: _itoa_word (in /usr/lib64/libc-2.22.so)
==28469==    by 0x4E8262F: vfprintf (in /usr/lib64/libc-2.22.so)
==28469==    by 0x4E88D48: printf (in /usr/lib64/libc-2.22.so)
==28469==    by 0x4005B9: main (in /home/alex/test)
...
...
...
```

实际上 `kmemcheck` 在内核空间做的事情，和 `valgrind` 在用户空间做的事情是一样的，都是用来检测未初始化的内存。

要想在内核中启用该机制，需要在配置内核时开启 `CONFIG_KMEMCHECK` 选项：

```
Kernel hacking
  -> Memory Debugging
```



`kmemcheck` 机制还提供了一些内核配置参数，我们可以在下一个段落中看到所有的可选参数。最后一个需要注意的是，`kmemcheck` 仅在 x86_64 体系中实现了。为了确信这一点，我们可以查看 `x86` 的内核配置文件 arch/x86/Kconfig：

```
config X86
  ...
  ...
  ...
  select HAVE_ARCH_KMEMCHECK
  ...
  ...
  ...
```

因此，对于其他的体系结构来说是没有 `kmemcheck` 功能的。

现在我们知道了 `kmemcheck` 可以检测内核中 `未初始化内存` 的使用情况，也知道了如何开启这个功能。那么 `kmemcheck` 是怎么做检测的呢？当内核尝试分配内存时，例如如下一段代码：

```
struct my_struct *my_struct = kmalloc(sizeof(struct my_struct), GFP_KERNEL);
```

或者换句话说，在内核访问 page 时会发生缺页中断。这是由于 `kmemcheck` 将内存页标记为 `不存在` （关于Linux内存分页的相关信息，你可以参考分页）。如果一个 `缺页中断` 异常发生了，异常处理程序会来处理这个异常，如果异常处理程序检测到内核使能了 `kmemcheck` ，那么就会将控制权提交给 `kmemcheck` 来处理；`kmemcheck` 检查完之后，该内存页会被标记为

`present` ，然后被中断的程序得以继续执行下去。 这里的处理方式比较巧妙，被中断程序的第一条指令执行时， `kmemcheck` 又会标记内存页为 `not present` ，按照这种方式，下一个对内存页的访问也会被捕获。

目前我们只是从理论层面考察了 `kmemcheck` ，接下来我们看一下Linux内核是怎么来实现该机制的。

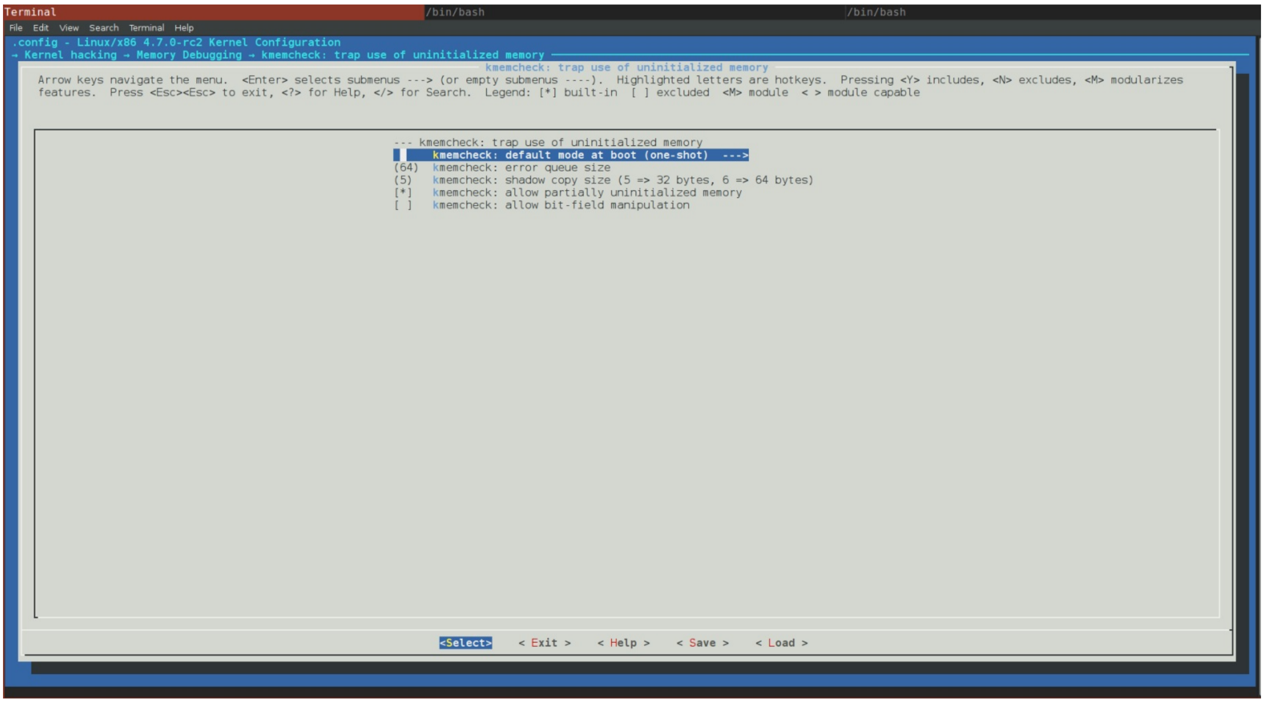
`kmemcheck` 机制在Linux内核中的实现

我们应该已经了解 `kmemcheck` 是做什么的以及它在Linux内核中的功能，现在是时候看一下它在Linux内核中的实现。 `kmemcheck` 在内核的实现分为两部分。第一部分是架构无关的部分，位于源码 mm/kmemcheck.c ；第二部分 x86_64架构相关的部分位于目录arch/x86/mm/kmemcheck中。

我们先分析该机制的初始化过程。我们已经知道要在内核中使能 `kmemcheck` 机制，需要开启内核的 `CONFIG_KMEMCHECK` 配置项。除了这个选项，我们还需要给内核command line传递一个 `kmemcheck` 参数：

- kmemcheck=0 (disabled)
- kmemcheck=1 (enabled)
- kmemcheck=2 (one-shot mode)

前面两个值得含义很明确，但是最后一个需要解释。这个选项会使 `kmemcheck` 进入一种特殊的模式：在第一次检测到未初始化内存的使用之后，就会关闭 `kmemcheck` 。实际上该模式是内核的默认选项：



从Linux初始化过程章节的第七节 part 中，我们知道在内核初始化过程中，会在 `do_initcall_level` , `do_early_param` 等函数中解析内核 command line。前面也提到过 `kmemcheck` 子系统由两部分组成，第一部分启动比较早。在源码 mm/kmemcheck.c 中有一个函数 `param_kmemcheck` ，该函数在command line解析时就会用到:

```
static int __init param_kmemcheck(char *str)
{
    int val;
    int ret;

    if (!str)
        return -EINVAL;

    ret = kstrtoint(str, 0, &val);
    if (ret)
        return ret;
    kmemcheck_enabled = val;
    return 0;
}

early_param("kmemcheck", param_kmemcheck);
```

从前面的介绍我们知道 `param_kmemcheck` 可能存在三种情况： `0` (使能), `1` (禁止) or `2` (一次性)。 `param_kmemcheck` 的实现很简单：将command line传递的 `kmemcheck` 参数的值由字符串转换为整数，然后赋值给变量 `kmemcheck_enabled` 。

第二阶段在内核初始化阶段执行，而不是在早期初始化过程 initcalls 。第二阶断的过程体现在 `kmemcheck_init` :

```
int __init kmemcheck_init(void)
{
    ...
    ...
    ...
}

early_initcall(kmemcheck_init);
```

`kmemcheck_init` 的主要目的就是调用 `kmemcheck_selftest` 函数，并检查它的返回值：

```
if (!kmemcheck_selftest()) {
    printk(KERN_INFO "kmemcheck: self-tests failed; disabling\n");
    kmemcheck_enabled = 0;
    return -EINVAL;
}

printk(KERN_INFO "kmemcheck: Initialized\n");
```

如果 `kmemcheck_init` 检测失败，就返回 `EINVAL` 。 `kmemcheck_selftest` 函数会检测内存访问相关的操作码（例如 `rep movsb`， `movzwq`)的大小。如果检测到的大小的实际大小是一致的， `kmemcheck_selftest` 返回 `true` ，否则返回 `false` 。

如果如下代码被调用:

```
struct my_struct *my_struct = kmalloc(sizeof(struct my_struct), GFP_KERNEL);
```

经过一系列的函数调用， `kmem_getpages` 函数会被调用到，该函数的定义在源码 mm/slab.c 中，该函数的主要功能就是尝试按照指定的参数需求分配内存页。在该函数的结尾处有如下代码：

```
if (kmemcheck_enabled && !(cachep->flags & SLAB_NOTRACK)) {
    kmemcheck_alloc_shadow(page, cachep->gfporder, flags, nodeid);

    if (cachep->ctor)
        kmemcheck_mark_uninitialized_pages(page, nr_pages);
    else
        kmemcheck_mark_unallocated_pages(page, nr_pages);
}
```

这段代码判断如果 `kmemcheck` 使能，并且参数中未设置 `SLAB_NOTRACK` ，那么就给分配的内存页设置 `non-present` 标记。 `SLAB_NOTRACK` 标记的含义是不跟踪未初始化的内存。另外，如果缓存对象有构造函数（细节在下面描述），所分配的内存页标记为未初始化，否则标记为未分配。 `kmemcheck_alloc_shadow` 函数在源码 mm/kmemcheck.c 中，其基本内容如下：

```
void kmemcheck_alloc_shadow(struct page *page, int order, gfp_t flags, int node)
{
    struct page *shadow;

       shadow = alloc_pages_node(node, flags | __GFP_NOTRACK, order);

       for(i = 0; i < pages; ++i)
        page[i].shadow = page_address(&shadow[i]);

       kmemcheck_hide_pages(page, pages);
}
```

首先为 shadow bits 分配内存，并为内存页设置 shadow 位。如果内存页设置了该标记，就意味着 `kmemcheck` 会跟踪这个内存页。最后调用 `kmemcheck_hide_pages` 函数。 `kmemcheck_hide_pages` 是体系结构相关的函数，其代码在 arch/x86/mm/kmemcheck/kmemcheck.c 源码中。该函数的功能是为指定的内存页设置 `non-present` 标记。该函数实现如下：

```
void kmemcheck_hide_pages(struct page *p, unsigned int n)
{
    unsigned int i;

    for (i = 0; i < n; ++i) {
        unsigned long address;
        pte_t *pte;
        unsigned int level;

        address = (unsigned long) page_address(&p[i]);
        pte = lookup_address(address, &level);
        BUG_ON(!pte);
        BUG_ON(level != PG_LEVEL_4K);

        set_pte(pte, __pte(pte_val(*pte) & ~_PAGE_PRESENT));
        set_pte(pte, __pte(pte_val(*pte) | _PAGE_HIDDEN));
        __flush_tlb_one(address);
    }
}
```

该函数遍历参数代表的所有内存页，并尝试获取每个内存页的 `页表项` 。如果获取成功，清理页表项的present 标记，设置页表项的 hidden 标记。在最后还需要刷新 TLB ,因为有一些内存页已经发生了改变。从这个地方开始，内存页就进入 `kmemcheck` 的跟踪系统。由于内存页的 `present` 标记被清除了，一旦 `kmalloc` 返回了内存地址，并且有代码访问这个地址，就会触发缺页中断。

在Linux内核初始化的第二节介绍过， `缺页中断` 处理程序是 arch/x86/mm/fault.c 的 `do_page_fault` 函数。该函数开始部分如下：

```
static noinline void
__do_page_fault(struct pt_regs *regs, unsigned long error_code,
        unsigned long address)
{
    ...
    ...
    ...
    if (kmemcheck_active(regs))
        kmemcheck_hide(regs);
    ...
    ...
    ...
}
```

`kmemcheck_active` 函数获取 `kmemcheck_context` per-cpu 结构体，并返回该结构体成员 `balance` 和0的比较结果：

```
bool kmemcheck_active(struct pt_regs *regs)
{
    struct kmemcheck_context *data = this_cpu_ptr(&kmemcheck_context);

    return data->balance > 0;
}
```

`kmemcheck_context` 结构体代表 `kmemcheck` 机制的当前状态。其内部保存了未初始化的地址，地址的数量等信息。其成员 `balance` 代表了 `kmemcheck` 的当前状态，换句话说，`balance` 表示 `kmemcheck` 是否已经隐藏了内存页。如果 `data->balance` 大于0，`kmemcheck_hide` 函数会被调用。这意味着 `kmemecheck` 已经设置了内存页的 `present` 标记，但是我们需要再次隐藏内存页以便触发下一次的缺页中断。 `kmemcheck_hide` 函数会清理内存页的 `present` 标记，这表示一次 `kmemcheck` 会话已经完成，新的缺页中断会再次被触发。在第一步，由于 `data->balance` 值为0，所以 `kmemcheck_active` 会返回false，所以 `kmemcheck_hide` 也不会被调用。接下来，我们看 `do_page_fault` 的下一行代码：

```
if (kmemcheck_fault(regs, address, error_code))
        return;
```

首先 `kmemcheck_fault` 函数检查引起错误的真实原因。第一步先检查标记寄存器以确认进程是否处于正常的内核态：

```
if (regs->flags & X86_VM_MASK)
        return false;
if (regs->cs != __KERNEL_CS)
        return false;
```

如果检测失败，表明这不是 `kmemcheck` 相关的缺页中断， `kmemcheck_fault` 会返回false。如果检测成功，接下来查找发生异常的地址的 `页表项` ，如果找不到页表项，函数返回false:

```
pte = kmemcheck_pte_lookup(address);
if (!pte)
    return false;
```

`kmemcheck_fault` 最后一步是调用 `kmemcheck_access` 函数，该函数检查对指定内存页的访问，并设置该内存页的present标记。 `kmemcheck_access` 函数做了大部分工作，它检查引起缺页异常的当前指令，如果检查到了错误，那么会把该错误的上下文保存到环形队列中：

```
static struct kmemcheck_error error_fifo[CONFIG_KMEMCHECK_QUEUE_SIZE];
```

`kmemcheck` 声明了一个特殊的 tasklet :

```
static DECLARE_TASKLET(kmemcheck_tasklet, &do_wakeup, 0);
```

该tasklet被调度执行时，会调用 `do_wakeup` 函数，该函数位于 arch/x86/mm/kmemcheck/error.c 文件中。

`do_wakeup` 函数调用 `kmemcheck_error_recall` 函数以便将 `kmemcheck` 检测到的错误信息输出。

```
kmemcheck_show(regs);
```

`kmemcheck_fault` 函数结束时会调用 `kmemcheck_show` 函数，该函数会再次设置内存页的present标记。

```
if (unlikely(data->balance != 0)) {
    kmemcheck_show_all();
    kmemcheck_error_save_bug(regs);
    data->balance = 0;
    return;
}
```

`kmemcheck_show_all` 函数会针对每个地址调用 `kmemcheck_show_addr` ：

```
static unsigned int kmemcheck_show_all(void)
{
    struct kmemcheck_context *data = this_cpu_ptr(&kmemcheck_context);
    unsigned int i;
    unsigned int n;

    n = 0;
    for (i = 0; i < data->n_addrs; ++i)
        n += kmemcheck_show_addr(data->addr[i]);

    return n;
}
```

`kmemcheck_show_addr` 函数内容如下:

```
int kmemcheck_show_addr(unsigned long address)
{
    pte_t *pte;

    pte = kmemcheck_pte_lookup(address);
    if (!pte)
        return 0;

    set_pte(pte, __pte(pte_val(*pte) | _PAGE_PRESENT));
    __flush_tlb_one(address);
    return 1;
}
```

在函数 `kmemcheck_show` 的结尾处会设置 TF 标记：

```
if (!(regs->flags & X86_EFLAGS_TF))
    data->flags = regs->flags;
```

我们之所以这么处理，是因为我们在内存页的缺页中断处理完后需要再次隐藏内存页。当 `TF` 标记被设置后，处理器在执行被中断程序的第一条指令时会进入单步模式，这会触发 `debug` 异常。从这个地方开始，内存页会被隐藏起来，执行流程继续。由于内存页不可见，那么访问内存页的时候又会触发缺页中断，然后 `kmemcheck` 就有机会继续检测/收集并显示内存错误信息。

到这里 `kmemcheck` 的工作机制就介绍完毕了。

结束语

Linux内核内存管理第三节介绍到此为止。如果你有任何疑问或者建议，你可以直接给我0xAX发消息， 发邮件，或者创建一个 issue 。 在接下来的小节中，我们来看一下另一个内存调试工具 - `kmemleak` 。

英文不是我的母语。如果你发现我的英文描述有任何问题，请提交一个**PR**到 **linux-insides.**

Links

- memory management
- debugging
- memory leaks
- kmemcheck documentation
- valgrind
- page fault

- initcalls
- opcode
- translation lookaside buffer
- per-cpu variables
- flags register
- tasklet
- Paging
- Previous part

控制组

这个章节描述了 Linux 内核中的控制组机制。

Control Groups

Introduction

This is the first part of the new chapter of the linux insides book and as you may guess by part's name - this part will cover control groups or `cgroups` mechanism in the Linux kernel.

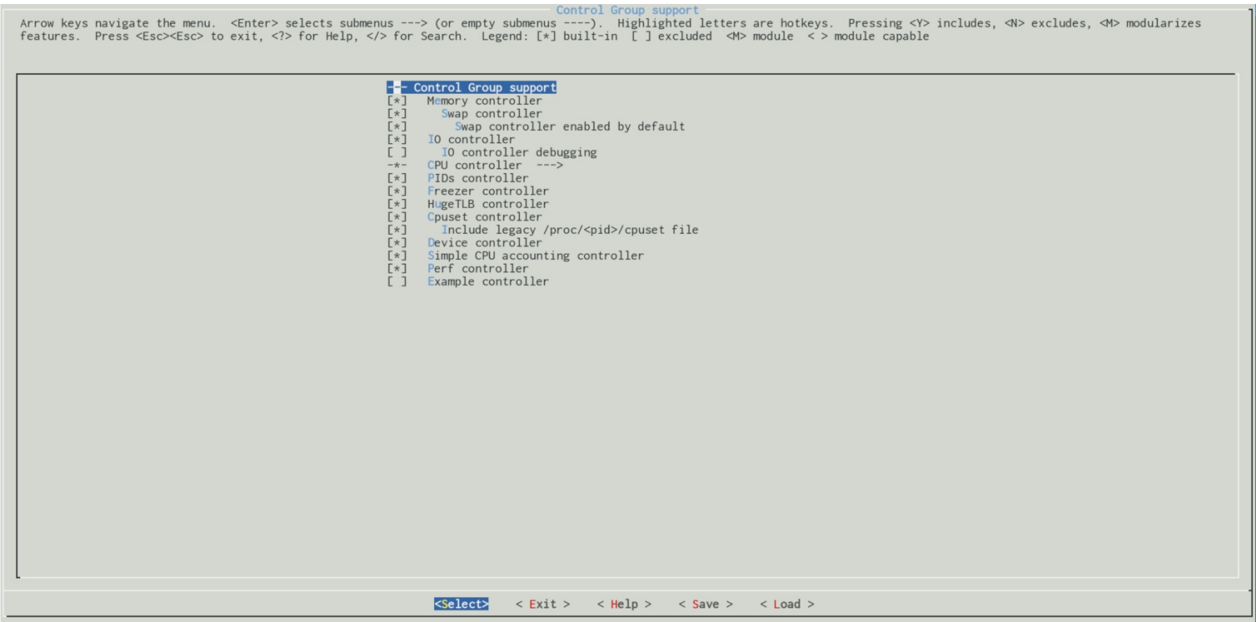
`Cgroups` are special mechanism provided by the Linux kernel which allows us to allocate kind of `resources` like processor time, number of processes per group, amount of memory per control group or combination of such resources for a process or set of processes. `Cgroups` are organized hierarchically and here this mechanism is similar to usual processes as they are hierarchical too and child `cgroups` inherit set of certain parameters from their parents. But actually they are not the same. The main differences between `cgroups` and normal processes that many different hierarchies of control groups may exist simultaneously in one time while normal process tree is always single. This was not a casual step because each control group hierarchy is attached to set of control group `subsystems` .

One `control group subsystem` represents one kind of resources like a processor time or number of pids or in other words number of processes for a `control group` . Linux kernel provides support for following twelve `control group subsystems` :

- `cpuset` - assigns individual processor(s) and memory nodes to task(s) in a group;
- `cpu` - uses the scheduler to provide cgroup tasks access to the processor resources;
- `cpuacct` - generates reports about processor usage by a group;
- `io` - sets limit to read/write from/to block devices;
- `memory` - sets limit on memory usage by a task(s) from a group;
- `devices` - allows access to devices by a task(s) from a group;
- `freezer` - allows to suspend/resume for a task(s) from a group;
- `net_cls` - allows to mark network packets from task(s) from a group;
- `net_prio` - provides a way to dynamically set the priority of network traffic per network interface for a group;
- `perf_event` - provides access to perf events) to a group;
- `hugetlb` - activates support for huge pages for a group;
- `pid` - sets limit to number of processes in a group.

Each of these control group subsystems depends on related configuration option. For example the `cpuset` subsystem should be enabled via `CONFIG_CPUSETS` kernel configuration option, the `io` subsystem via `CONFIG_BLK_CGROUP` kernel configuration option and etc. All of

these kernel configuration options may be found in the `General setup → Control Group support` menu:



You may see enabled control groups on your computer via proc filesystem:

```
$ cat /proc/cgroups
#subsys_name    hierarchy    num_cgroups    enabled
cpuset    8    1    1
cpu    7    66    1
cpuacct    7    66    1
blkio    11    66    1
memory    9    94    1
devices    6    66    1
freezer    2    1    1
net_cls    4    1    1
perf_event    3    1    1
net_prio    4    1    1
hugetlb    10    1    1
pids    5    69    1
```

or via sysfs:

```
$ ls -l /sys/fs/cgroup/
total 0
dr-xr-xr-x 5 root root  0 Dec  2 22:37 blkio
lrwxrwxrwx 1 root root 11 Dec  2 22:37 cpu -> cpu,cpuacct
lrwxrwxrwx 1 root root 11 Dec  2 22:37 cpuacct -> cpu,cpuacct
dr-xr-xr-x 5 root root  0 Dec  2 22:37 cpu,cpuacct
dr-xr-xr-x 2 root root  0 Dec  2 22:37 cpuset
dr-xr-xr-x 5 root root  0 Dec  2 22:37 devices
dr-xr-xr-x 2 root root  0 Dec  2 22:37 freezer
dr-xr-xr-x 2 root root  0 Dec  2 22:37 hugetlb
dr-xr-xr-x 5 root root  0 Dec  2 22:37 memory
lrwxrwxrwx 1 root root 16 Dec  2 22:37 net_cls -> net_cls,net_prio
dr-xr-xr-x 2 root root  0 Dec  2 22:37 net_cls,net_prio
lrwxrwxrwx 1 root root 16 Dec  2 22:37 net_prio -> net_cls,net_prio
dr-xr-xr-x 2 root root  0 Dec  2 22:37 perf_event
dr-xr-xr-x 5 root root  0 Dec  2 22:37 pids
dr-xr-xr-x 5 root root  0 Dec  2 22:37 systemd
```

As you already may guess that `control groups` mechanism is not such mechanism which was invented only directly to the needs of the Linux kernel, but mostly for userspace needs. To use a `control group`, we should create it at first. We may create a `cgroup` via two ways.

The first way is to create subdirectory in any subsystem from `/sys/fs/cgroup` and add a pid of a task to a `tasks` file which will be created automatically right after we will create the subdirectory.

The second way is to create/destroy/manage `cgroups` with utils from `libcgroup` library (`libcgroup-tools` in Fedora).

Let's consider simple example. Following bash script will print a line to `/dev/tty` device which represents control terminal for the current process:

```
#!/bin/bash

while :
do
    echo "print line" > /dev/tty
    sleep 5
done
```

So, if we will run this script we will see following result:

```
$ sudo chmod +x cgroup_test_script.sh
~$ ./cgroup_test_script.sh
print line
print line
print line
...
...
...
```

Now let's go to the place where `cgroupfs` is mounted on our computer. As we just saw, this is `/sys/fs/cgroup` directory, but you may mount it everywhere you want.

```
$ cd /sys/fs/cgroup
```

And now let's go to the `devices` subdirectory which represents kind of resources that allows or denies access to devices by tasks in a `cgroup` :

```
# cd devices
```

and create `cgroup_test_group` directory there:

```
# mkdir cgroup_test_group
```

After creation of the `cgroup_test_group` directory, following files will be generated there:

```
/sys/fs/cgroup/devices/cgroup_test_group$ ls -l
total 0
-rw-r--r-- 1 root root 0 Dec  3 22:55 cgroup.clone_children
-rw-r--r-- 1 root root 0 Dec  3 22:55 cgroup.procs
--w------- 1 root root 0 Dec  3 22:55 devices.allow
--w------- 1 root root 0 Dec  3 22:55 devices.deny
-r--r--r-- 1 root root 0 Dec  3 22:55 devices.list
-rw-r--r-- 1 root root 0 Dec  3 22:55 notify_on_release
-rw-r--r-- 1 root root 0 Dec  3 22:55 tasks
```

For this moment we are interested in `tasks` and `devices.deny` files. The first `tasks` files should contain pid(s) of processes which will be attached to the `cgroup_test_group` . The second `devices.deny` file contain list of denied devices. By default a newly created group has no any limits for devices access. To forbid a device (in our case it is `/dev/tty`) we should write to the `devices.deny` following line:

```
# echo "c 5:0 w" > devices.deny
```

Let's go step by step through this line. The first `c` letter represents type of a device. In our case the `/dev/tty` is `char device`. We can verify this from output of `ls` command:

```
~$ ls -l /dev/tty
crw-rw-rw- 1 root tty 5, 0 Dec  3 22:48 /dev/tty
```

see the first `c` letter in a permissions list. The second part is `5:0` is minor and major numbers of the device. You can see these numbers in the output of `ls` too. And the last `w` letter forbids tasks to write to the specified device. So let's start the `cgroup_test_script.sh` script:

```
~$ ./cgroup_test_script.sh
print line
print line
print line
...
...
```

and add pid of this process to the `devices/tasks` file of our group:

```
# echo $(pidof -x cgroup_test_script.sh) > /sys/fs/cgroup/devices/cgroup_test_group/ta
sks
```

The result of this action will be as expected:

```
~$ ./cgroup_test_script.sh
print line
print line
print line
print line
print line
print line
./cgroup_test_script.sh: line 5: /dev/tty: Operation not permitted
```

Similar situation will be when you will run you docker) containers for example:

```
~$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             S
TATUS              PORTS                    NAMES
fa2d2085cd1c        mariadb:10          "docker-entrypoint..."   12 days ago         U
p 4 minutes        0.0.0.0:3306->3306/tcp   mysql-work

~$ cat /sys/fs/cgroup/devices/docker/fa2d2085cd1c8d797002c77387d2061f56fefb470892f140d
0dc511bd4d9bb61/tasks | head -3
5501
5584
5585
...
...
...
```

So, during startup of a `docker` container, `docker` will create a `cgroup` for processes in this container:

```
$ docker exec -it mysql-work /bin/bash
$ top
 PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND                                                                     1 mysql     20
  0  963996 101268  15744 S   0.0  0.6   0:00.46 mysqld
                                                       71 root      20   0   20248   3028
  2732 S   0.0  0.0   0:00.01 bash
                                   77 root      20   0   21948   2424   2056 R   0.0  0.0
   0:00.00 top
```

And we may see this `cgroup` on host machine:

```
$ systemd-cgls

Control group /:
-.slice
├─docker
│ └─fa2d2085cd1c8d797002c77387d2061f56fefb470892f140d0dc511bd4d9bb61
│   ├─5501 mysqld
│   └─6404 /bin/bash
```

Now we know a little about `control groups` mechanism, how to use it manually and what's purpose of this mechanism. Time to look inside of the Linux kernel source code and start to dive into implementation of this mechanism.

Early initialization of control groups

Now after we just saw little theory about `control groups` Linux kernel mechanism, we may start to dive into the source code of Linux kernel to acquainted with this mechanism closer. As always we will start from the initialization of `control groups`. Initialization of `cgroups` divided into two parts in the Linux kernel: early and late. In this part we will consider only `early` part and `late` part will be considered in next parts.

Early initialization of `cgroups` starts from the call of the:

```
cgroup_init_early();
```

function in the init/main.c during early initialization of the Linux kernel. This function is defined in the kernel/cgroup.c source code file and starts from the definition of two following local variables:

```
int __init cgroup_init_early(void)
{
    static struct cgroup_sb_opts __initdata opts;
    struct cgroup_subsys *ss;
    ...
    ...
    ...
}
```

The `cgroup_sb_opts` structure defined in the same source code file and looks:

```
struct cgroup_sb_opts {
    u16 subsys_mask;
    unsigned int flags;
    char *release_agent;
    bool cpuset_clone_children;
    char *name;
    bool none;
};
```

which represents mount options of `cgroupfs`. For example we may create named cgroup hierarchy (with name `my_cgrp`) with the `name=` option and without any subsystems:

```
$ mount -t cgroup -oname=my_cgrp,none /mnt/cgroups
```

The second variable - `ss` has type - `cgroup_subsys` structure which is defined in the include/linux/cgroup-defs.h header file and as you may guess from the name of the type, it represents a `cgroup` subsystem. This structure contains various fields and callback functions like:

```
struct cgroup_subsys {
    int (*css_online)(struct cgroup_subsys_state *css);
    void (*css_offline)(struct cgroup_subsys_state *css);
    ...
    ...
    ...
    bool early_init:1;
    int id;
    const char *name;
    struct cgroup_root *root;
    ...
    ...
    ...
}
```

Where for example `css_online` and `css_offline` callbacks are called after a cgroup successfully will complete all allocations and a cgroup will be before releasing respectively. The `early_init` flags marks subsystems which may/should be initialized early. The `id` and `name` fields represents unique identifier in the array of registered subsystems for a cgroup and `name` of a subsystem respectively. The last - `root` fields represents pointer to the root of of a cgroup hierarchy.

Of course the `cgroup_subsys` structure is bigger and has other fields, but it is enough for now. Now as we got to know important structures related to `cgroups` mechanism, let's return to the `cgroup_init_early` function. Main purpose of this function is to do early initialization of some subsystems. As you already may guess, these `early` subsystems should have `cgroup_subsys->early_init = 1`. Let's look what subsystems may be initialized early.

After the definition of the two local variables we may see following lines of code:

```
init_cgroup_root(&cgrp_dfl_root, &opts);
cgrp_dfl_root.cgrp.self.flags |= CSS_NO_REF;
```

Here we may see call of the `init_cgroup_root` function which will execute initialization of the default unified hierarchy and after this we set `CSS_NO_REF` flag in state of this default `cgroup` to disable reference counting for this css. The `cgrp_dfl_root` is defined in the same source code file:

```
struct cgroup_root cgrp_dfl_root;
```

Its `cgrp` field represented by the `cgroup` structure which represents a `cgroup` as you already may guess and defined in the include/linux/cgroup-defs.h header file. We already know that a process which is represented by the `task_struct` in the Linux kernel. The

`task_struct` does not contain direct link to a `cgroup` where this task is attached. But it may be reached via `css_set` field of the `task_struct`. This `css_set` structure holds pointer to the array of subsystem states:

```
struct css_set {
    ...
    ...
    ....
    struct cgroup_subsys_state *subsys[CGROUP_SUBSYS_COUNT];
    ...
    ...
    ...
}
```

And via the `cgroup_subsys_state`, a process may get a `cgroup` that this process is attached to:

```
struct cgroup_subsys_state {
    ...
    ...
    ...
    struct cgroup *cgroup;
    ...
    ...
    ...
}
```

So, the overall picture of `cgroups` related data structure is following:

```
+-------------+         +---------------------+    +------------->+-------------------
--+          +----------------+
| task_struct |          |       css_set       |     |               | cgroup_subsys_stat
e |          |     cgroup     |
+-------------+          |                     |     |               +-------------------
--+          +----------------+
|             |          |                     |     |               |
  |          |     flags      |
|             |          |                     |     |               +-------------------
--+          |  cgroup.procs  |
|             |          |                     |     |               |        cgroup
  |--------->|       id       |
|             |          |                     |     |               +-------------------
--+          |      ....      |
|-------------+          |---------------------+----+
             +----------------+
|   cgroups   | ------> | cgroup_subsys_state | array of cgroup_subsys_state
|-------------+         +---------------------+------------------>+-------------------
--+          +----------------+
```

```
 |             |         |                   |                 | cgroup_subsys_stat
e |           |      cgroup    |
+-------------+         +--------------------+                 +------------------
--+          +----------------+
                                                                |
  |          |      flags     |
                                                                +------------------
--+          |   cgroup.procs |
                                                                |        cgroup
  |--------->|        id      |
                                                                +------------------
--+          |       ....     |
                                                                |    cgroup_subsys
  |          +----------------+
                                                                +------------------
--+
                                                                          |
                                                                          |
                                                                          ↓
                                                                +------------------
--+
                                                                |    cgroup_subsys
  |
                                                                +------------------
--+
                                                                |         id
  |
                                                                |        name
  |
                                                                |      css_online
  |
                                                                |      css_ofline
  |
                                                                |        attach
  |
                                                                |         ....
  |
                                                                +------------------
--+
```

So, the `init_cgroup_root` fills the `cgrp_dfl_root` with the default values. The next thing is assigning initial `css_set` to the `init_task` which represents first process in the system:

```
RCU_INIT_POINTER(init_task.cgroups, &init_css_set);
```

And the last big thing in the `cgroup_init_early` function is initialization of `early cgroups`. Here we go over all registered subsystems and assign unique identity number, name of a subsystem and call the `cgroup_init_subsys` function for subsystems which are marked as early:

```
for_each_subsys(ss, i) {
        ss->id = i;
        ss->name = cgroup_subsys_name[i];

        if (ss->early_init)
            cgroup_init_subsys(ss, true);
}
```

The `for_each_subsys` here is a macro which is defined in the kernel/cgroup.c source code file and just expands to the `for` loop over `cgroup_subsys` array. Definition of this array may be found in the same source code file and it looks in a little unusual way:

```
#define SUBSYS(_x) [_x ## _cgrp_id] = &_x ## _cgrp_subsys,
    static struct cgroup_subsys *cgroup_subsys[] = {
        #include <linux/cgroup_subsys.h>
};
#undef SUBSYS
```

It is defined as `SUBSYS` macro which takes one argument (name of a subsystem) and defines `cgroup_subsys` array of cgroup subsystems. Additionally we may see that the array is initialized with content of the linux/cgroup_subsys.h header file. If we will look inside of this header file we will see again set of the `SUBSYS` macros with the given subsystems names:

```
#if IS_ENABLED(CONFIG_CPUSETS)
SUBSYS(cpuset)
#endif

#if IS_ENABLED(CONFIG_CGROUP_SCHED)
SUBSYS(cpu)
#endif
...
...
...
```

This works because of `#undef` statement after first definition of the `SUBSYS` macro. Look at the `&_x ## _cgrp_subsys` expression. The `##` operator concatenates right and left expression in a `C` macro. So as we passed `cpuset`, `cpu` and etc., to the `SUBSYS` macro, somewhere `cpuset_cgrp_subsys`, `cp_cgrp_subsys` should be defined. And that's true. If you will look in the kernel/cpuset.c source code file, you will see this definition:

```
struct cgroup_subsys cpuset_cgrp_subsys = {
    ...
    ...
    ...
    .early_init    = true,
};
```

So the last step in the `cgroup_init_early` function is initialization of early subsystems with the call of the `cgroup_init_subsys` function. Following early subsystems will be initialized:

- `cpuset` ;
- `cpu` ;
- `cpuacct` .

The `cgroup_init_subsys` function does initialization of the given subsystem with the default values. For example sets root of hierarchy, allocates space for the given subsystem with the call of the `css_alloc` callback function, link a subsystem with a parent if it exists, add allocated subsystem to the initial process and etc.

That's all. From this moment early subsystems are initialized.

Conclusion

It is the end of the first part which describes introduction into `Control groups` mechanism in the Linux kernel. We covered some theory and the first steps of initialization of stuffs related to `control groups` mechanism. In the next part we will continue to dive into the more practical aspects of `control groups` .

If you have any questions or suggestions write me a comment or ping me at twitter.

Please note that English is not my first language, And I am really sorry for any inconvenience. If you find any mistakes please send me a PR to linux-insides.

Links

- control groups
- PID
- cpuset
- block devices
- huge pages
- sysfs
- proc

- cgroups kernel documentation
- cgroups v2
- bash
- docker)
- perf events)
- Previous chapter

Linux 内核概念

本章描述内核中使用到的各种各样的概念。

Per-cpu 变量

Per-cpu 变量是一项内核特性。从它的名字你就可以理解这项特性的意义了。我们可以创建一个变量，然后每个 CPU 上都会有一个此变量的拷贝。本节我们来看下这个特性，并试着去理解它是如何实现以及工作的。

内核提供了一个创建 per-cpu 变量的 API - `DEFINE_PER_CPU` 宏：

```
#define DEFINE_PER_CPU(type, name) \
        DEFINE_PER_CPU_SECTION(type, name, "")
```

正如其它许多处理 per-cpu 变量的宏一样，这个宏定义在 include/linux/percpu-defs.h 中。现在我们来看下这个特性是如何实现的。

看下 `DECLARE_PER_CPU` 的定义，可以看到它使用了 2 个参数：`type` 和 `name`，因此我们可以这样创建 per-cpu 变量：

```
DEFINE_PER_CPU(int, per_cpu_n)
```

我们传入要创建变量的类型和名字，`DEFINE_PER_CPU` 调用 `DEFINE_PER_CPU_SECTION`，将两个参数和空字符串传递给后者。让我们来看下 `DEFINE_PER_CPU_SECTION` 的定义：

```
#define DEFINE_PER_CPU_SECTION(type, name, sec)    \
         __PCPU_ATTRS(sec) PER_CPU_DEF_ATTRIBUTES  \
         __typeof__(type) name
```

```
#define __PCPU_ATTRS(sec)                                                \
         __percpu __attribute__((section(PER_CPU_BASE_SECTION sec)))     \
         PER_CPU_ATTRIBUTES
```

其中 `section` 是:

```
#define PER_CPU_BASE_SECTION ".data..percpu"
```

当所有的宏展开之后，我们得到一个全局的 per-cpu 变量：

```
__attribute__((section(".data..percpu"))) int per_cpu_n
```

这意味着我们在 `.data..percpu` 段有了一个 `per_cpu_n` 变量，可以在 `vmlinux` 中找到它：

```
.data..percpu 00013a58  0000000000000000  0000000001a5c000  00e00000  2**12
              CONTENTS, ALLOC, LOAD, DATA
```

好，现在我们知道了，当我们使用 `DEFINE_PER_CPU` 宏时，一个在 `.data..percpu` 段中的 per-cpu 变量就被创建了。内核初始化时，调用 `setup_per_cpu_areas` 函数多次加载 `.data..percpu` 段，每个 CPU 一次。

让我们来看下 per-cpu 区域初始化流程。它从 init/main.c 中调用 `setup_per_cpu_areas` 函数开始，这个函数定义在 arch/x86/kernel/setup_percpu.c 中。

```
pr_info("NR_CPUS:%d nr_cpumask_bits:%d nr_cpu_ids:%d nr_node_ids:%d\n",
        NR_CPUS, nr_cpumask_bits, nr_cpu_ids, nr_node_ids);
```

`setup_per_cpu_areas` 开始输出在内核配置中以 `CONFIG_NR_CPUS` 配置项设置的最大 CPUs 数，实际的 CPU 个数，`nr_cpumask_bits`（对于新的 `cpumask` 操作来说和 `NR_CPUS` 是一样的），还有 `NUMA` 节点个数。

我们可以在 `dmesg` 中看到这些输出：

```
$ dmesg | grep percpu
[    0.000000] setup_percpu: NR_CPUS:8 nr_cpumask_bits:8 nr_cpu_ids:8 nr_node_ids:1
```

然后我们检查 `per-cpu` 第一个块分配器。所有的 per-cpu 区域都是以块进行分配的。第一个块用于静态 per-cpu 变量。Linux 内核提供了决定第一个块分配器类型的命令行：`percpu_alloc`。我们可以在内核文档中读到它的说明。

```
percpu_alloc=    选择要使用哪个 per-cpu 第一个块分配器。
        当前支持的类型是 "embed" 和 "page"。
        不同架构支持这些类型的子集或不支持。
        更多分配器的细节参考 mm/percpu.c 中的注释。
        这个参数主要是为了调试和性能比较的。
```

mm/percpu.c 包含了这个命令行选项的处理函数：

```
early_param("percpu_alloc", percpu_alloc_setup);
```

其中 `percpu_alloc_setup` 函数根据 `percpu_alloc` 参数值设置 `pcpu_chosen_fc` 变量。默认第一个块分配器是 `auto`：

```
enum pcpu_fc pcpu_chosen_fc __initdata = PCPU_FC_AUTO;
```

如果内核命令行中没有设置 `percpu_alloc` 参数，就会使用 `embed` 分配器，将第一个 per-cpu 块嵌入进带 memblock 的 bootmem。最后一个分配器和第一个块 `page` 分配器一样，只是将第一个块使用 `PAGE_SIZE` 页进行了映射。

如我上面所写，首先我们在 `setup_per_cpu_areas` 中对第一个块分配器检查，检查到第一个块分配器不是 page 分配器：

```
if (pcpu_chosen_fc != PCPU_FC_PAGE) {
    ...
    ...
    ...
}
```

如果不是 `PCPU_FC_PAGE`，我们就使用 `embed` 分配器并使用 `pcpu_embed_first_chunk` 函数分配第一块空间。

```
rc = pcpu_embed_first_chunk(PERCPU_FIRST_CHUNK_RESERVE,
                        dyn_size, atom_size,
                        pcpu_cpu_distance,
                        pcpu_fc_alloc, pcpu_fc_free);
```

如前所述，函数 `pcpu_embed_first_chunk` 将第一个 per-cpu 块嵌入 bootmen，因此我们传递一些参数给 `pcpu_embed_first_chunk`。参数如下：

- `PERCPU_FIRST_CHUNK_RESERVE` - 为静态变量 `per-cpu` 保留空间的大小；
- `dyn_size` - 动态分配的最少空闲字节；
- `atom_size` - 所有的分配都是这个的整数倍，并以此对齐；
- `pcpu_cpu_distance` - 决定 cpus 距离的回调函数；
- `pcpu_fc_alloc` - 分配 `percpu` 页的函数；
- `pcpu_fc_free` - 释放 `percpu` 页的函数。

在调用 `pcpu_embed_first_chunk` 前我们计算好所有的参数：

```
const size_t dyn_size = PERCPU_MODULE_RESERVE + PERCPU_DYNAMIC_RESERVE - PERCPU_FIRST_
CHUNK_RESERVE;
size_t atom_size;
#ifdef CONFIG_X86_64
        atom_size = PMD_SIZE;
#else
        atom_size = PAGE_SIZE;
#endif
```

如果第一个块分配器是 `PCPU_FC_PAGE`，我们用 `pcpu_page_first_chunk` 而不是 `pcpu_embed_first_chunk`。 `per-cpu` 区域准备好以后，我们用 `setup_percpu_segment` 函数设置 `per-cpu` 的偏移和段（只针对 `x86` 系统），并将前面的数据从数组移到 `per-cpu` 变量

（ `x86_cpu_to_apicid` , `irq_stack_ptr` 等等）。当内核完成初始化进程后，我们就有了N个 `.data..percpu` 段，其中 N 是 CPU 个数，bootstrap 进程使用的段将会包含用 `DEFINE_PER_CPU` 宏创建的未初始化的变量。

内核提供了操作 per-cpu 变量的API：

- get_cpu_var(var)
- put_cpu_var(var)

让我们来看看 `get_cpu_var` 的实现：

```
#define get_cpu_var(var)     \
(*({                         \
         preempt_disable();  \
         this_cpu_ptr(&var); \
}))
```

Linux 内核是抢占式的，获取 per-cpu 变量需要我们知道内核运行在哪个处理器上。因此访问 per-cpu 变量时，当前代码不能被抢占，不能移到其它的 CPU。如我们所见，这就是为什么首先调用 `preempt_disable` 函数然后调用 `this_cpu_ptr` 宏，像这样：

```
#define this_cpu_ptr(ptr) raw_cpu_ptr(ptr)
```

以及

```
#define raw_cpu_ptr(ptr)        per_cpu_ptr(ptr, 0)
```

`per_cpu_ptr` 返回一个指向给定 CPU（第 2 个参数） per-cpu 变量的指针。当我们创建了一个 per-cpu 变量并对其进行了修改时，我们必须调用 `put_cpu_var` 宏通过函数 `preempt_enable` 使能抢占。因此典型的 per-cpu 变量的使用如下：

```
get_cpu_var(var);
...
//用这个 'var' 做些啥
...
put_cpu_var(var);
```

让我们来看下这个 `per_cpu_ptr` 宏：

```
#define per_cpu_ptr(ptr, cpu)                             \
({                                                        \
        __verify_pcpu_ptr(ptr);                           \
         SHIFT_PERCPU_PTR((ptr), per_cpu_offset((cpu)));  \
})
```

就像我们上面写的，这个宏返回了一个给定 cpu 的 per-cpu 变量。首先它调用了 `__verify_pcpu_ptr`：

```
#define __verify_pcpu_ptr(ptr)
do {
    const void __percpu *__vpp_verify = (typeof((ptr) + 0))NULL;
    (void)__vpp_verify;
} while (0)
```

该宏声明了 `ptr` 类型的 `const void __percpu *`。

之后，我们可以看到带两个参数的 `SHIFT_PERCPU_PTR` 宏的调用。第一个参数是我们的指针，第二个参数是传给 `per_cpu_offset` 宏的CPU数：

```
#define per_cpu_offset(x) (__per_cpu_offset[x])
```

该宏将 `x` 扩展为 `__per_cpu_offset` 数组：

```
extern unsigned long __per_cpu_offset[NR_CPUS];
```

其中 `NR_CPUS` 是 CPU 的数目。`__per_cpu_offset` 数组以 CPU 变量拷贝之间的距离填充。例如，所有 per-cpu 变量是 `X` 字节大小，所以我们通过 `__per_cpu_offset[Y]` 就可以访问 `X*Y`。让我们来看下 `SHIFT_PERCPU_PTR` 的实现：

```
#define SHIFT_PERCPU_PTR(__p, __offset)                                 \
         RELOC_HIDE((typeof(*(__p)) __kernel __force *)(__p), (__offset))
```

`RELOC_HIDE` 只是取得偏移量 `(typeof(ptr)) (__ptr + (off))`，并返回一个指向该变量的指针。

就这些了！当然这不是全部的 API，只是一个大概。开头是比较艰难，但是理解 per-cpu 变量你只需理解 include/linux/percpu-defs.h 的奥秘。

让我们再看下获得 per-cpu 变量指针的算法：

- 内核在初始化流程中创建多个 `.data..percpu` 段（一个 per-cpu 变量一个）；
- 所有 `DEFINE_PER_CPU` 宏创建的变量都将重新分配到首个扇区或者 CPU0；
- `__per_cpu_offset` 数组以 (`BOOT_PERCPU_OFFSET`) 和 `.data..percpu` 扇区之间的距离填充；
- 当 `per_cpu_ptr` 被调用时，例如取一个 per-cpu 变量的第三个 CPU 的指针，将访问 `__per_cpu_offset` 数组，该数组的索引指向了所需 CPU。

就这么多了。

CPU masks

介绍

`Cpumasks` 是Linux内核提供的保存系统CPU信息的特殊方法。包含 `Cpumasks` 操作 API 相关的源码和头文件：

- include/linux/cpumask.h
- lib/cpumask.c
- kernel/cpu.c

正如 include/linux/cpumask.h 注释：Cpumasks 提供了代表系统中 CPU 集合的位图，一位放置一个 CPU 序号。我们已经在 Kernel entry point 部分，函数 `boot_cpu_init` 中看到了一点 cpumask。这个函数将第一个启动的 cpu 上线、激活等等……

```
set_cpu_online(cpu, true);
set_cpu_active(cpu, true);
set_cpu_present(cpu, true);
set_cpu_possible(cpu, true);
```

`set_cpu_possible` 是一个在系统启动时任意时刻都可插入的 cpu ID 集合。`cpu_present` 代表了当前插入的 CPUs。`cpu_online` 是 `cpu_present` 的子集，表示可调度的 CPUs。这些掩码依赖于 `CONFIG_HOTPLUG_CPU` 配置选项，以及 `possible == present` 和 `active == online` 选项是否被禁用。这些函数的实现很相似，检测第二个参数，如果为 `true`，就调用 `cpumask_set_cpu`，否则调用 `cpumask_clear_cpu`。

有两种方法创建 `cpumask`。第一种是用 `cpumask_t`。定义如下：

```
typedef struct cpumask { DECLARE_BITMAP(bits, NR_CPUS); } cpumask_t;
```

它封装了 `cpumask` 结构，其包含了一个位掩码 `bits` 字段。`DECLARE_BITMAP` 宏有两个参数：

- bitmap name;
- number of bits.

并以给定名称创建了一个 `unsigned long` 数组。它的实现非常简单：

```
#define DECLARE_BITMAP(name,bits) \
        unsigned long name[BITS_TO_LONGS(bits)]
```

其中 `BITS_TO_LONGS` ：

```
#define BITS_TO_LONGS(nr)       DIV_ROUND_UP(nr, BITS_PER_BYTE * sizeof(long))
#define DIV_ROUND_UP(n,d) (((n) + (d) - 1) / (d))
```

因为我们专注于 `x86_64` 架构， `unsigned long` 是8字节大小，因此我们的数组仅包含一个元素：

```
(((8) + (8) - 1) / (8)) = 1
```

`NR_CPUS` 宏表示的是系统中 CPU 的数目，且依赖于在 include/linux/threads.h 中定义的 `CONFIG_NR_CPUS` 宏，看起来像这样：

```
#ifndef CONFIG_NR_CPUS
        #define CONFIG_NR_CPUS  1
#endif

#define NR_CPUS         CONFIG_NR_CPUS
```

第二种定义 cpumask 的方法是直接使用宏 `DECLARE_BITMAP` 和 `to_cpumask` 宏，后者将给定的位图转化为 `struct cpumask *` ：

```
#define to_cpumask(bitmap)                                              \
        ((struct cpumask *)(1 ? (bitmap)                                \
                            : (void *)sizeof(__check_is_bitmap(bitmap))))
```

可以看到这里的三目运算符每次总是 `true` 。 `__check_is_bitmap` 内联函数定义为：

```
static inline int __check_is_bitmap(const unsigned long *bitmap)
{
        return 1;
}
```

每次都是返回 `1` 。我们需要它只是因为：编译时检测一个给定的 `bitmap` 是一个位图，换句话说，它检测一个 `bitmap` 是否有 `unsigned long *` 类型。因此我们传递 `cpu_possible_bits` 给宏 `to_cpumask` ，将 `unsigned long` 数组转换为 `struct cpumask *` 。

cpumask API

因为我们可以用其中一个方法来定义 cpumask，Linux 内核提供了 API 来处理 cpumask。我们来研究下其中一个函数，例如 `set_cpu_online` ，这个函数有两个参数：

* CPU 数目;
* CPU 状态;

这个函数的实现如下所示：

```
void set_cpu_online(unsigned int cpu, bool online)
{
    if (online) {
        cpumask_set_cpu(cpu, to_cpumask(cpu_online_bits));
        cpumask_set_cpu(cpu, to_cpumask(cpu_active_bits));
    } else {
        cpumask_clear_cpu(cpu, to_cpumask(cpu_online_bits));
    }
}
```

该函数首先检测第二个 `state` 参数并调用依赖它的 `cpumask_set_cpu` 或 `cpumask_clear_cpu` 。这里我们可以看到在中 `cpumask_set_cpu` 的第二个参数转换为 `struct cpumask *` 。在我们的例子中是位图 `cpu_online_bits` ，定义如下：

```
static DECLARE_BITMAP(cpu_online_bits, CONFIG_NR_CPUS) __read_mostly;
```

函数 `cpumask_set_cpu` 仅调用了一次 `set_bit` 函数：

```
static inline void cpumask_set_cpu(unsigned int cpu, struct cpumask *dstp)
{
        set_bit(cpumask_check(cpu), cpumask_bits(dstp));
}
```

`set_bit` 函数也有两个参数，设置了一个给定位（第一个参数）的内存（第二个参数或 `cpu_online_bits` 位图）。这儿我们可以看到在调用 `set_bit` 之前，它的两个参数会传递给

* cpumask_check;
* cpumask_bits.

让我们细看下这两个宏。第一个 `cpumask_check` 在我们的例子里没做任何事，只是返回了给的参数。第二个 `cpumask_bits` 只是返回了传入 `struct cpumask *` 结构的 `bits` 域。

```
#define cpumask_bits(maskp) ((maskp)->bits)
```

现在让我们看下 `set_bit` 的实现：

```
static __always_inline void
set_bit(long nr, volatile unsigned long *addr)
{
        if (IS_IMMEDIATE(nr)) {
                asm volatile(LOCK_PREFIX "orb %1,%0"
                        : CONST_MASK_ADDR(nr, addr)
                        : "iq" ((u8)CONST_MASK(nr))
                        : "memory");
        } else {
                asm volatile(LOCK_PREFIX "bts %1,%0"
                        : BITOP_ADDR(addr) : "Ir" (nr) : "memory");
        }
}
```

这个函数看着吓人，但它没有看起来那么难。首先传参 `nr` 或者说位数给 `IS_IMMEDIATE` 宏，该宏调用了 GCC 内联函数 `__builtin_constant_p` ：

```
#define IS_IMMEDIATE(nr)    (__builtin_constant_p(nr))
```

`__builtin_constant_p` 检查给定参数是否编译时恒定变量。因为我们的 `cpu` 不是编译时恒定变量，将会执行 `else` 分支：

```
asm volatile(LOCK_PREFIX "bts %1,%0" : BITOP_ADDR(addr) : "Ir" (nr) : "memory");
```

让我们试着一步一步来理解它如何工作的：

`LOCK_PREFIX` 是个 x86 `lock` 指令。这个指令告诉 CPU 当指令执行时占据系统总线。这允许 CPU 同步内存访问，防止多核（或多设备 - 比如 DMA 控制器）并发访问同一个内存cell。

`BITOP_ADDR` 转换给定参数至 `(*(volatile long *)` 并且加了 `+m` 约束。 `+` 意味着这个操作数对于指令是可读写的。 `m` 显示这是一个内存操作数。 `BITOP_ADDR` 定义如下：

```
#define BITOP_ADDR(x) "+m" (*(volatile long *) (x))
```

接下来是 `memory` 。它告诉编译器汇编代码执行内存读或写到某些项，而不是那些输入或输出操作数（例如，访问指向输出参数的内存）。

`Ir` - 寄存器操作数。

`bts` 指令设置一个位字符串的给定位，存储给定位的值到 `CF` 标志位。所以我们传递 cpu 号，我们的例子中为 0，给 `set_bit` 并且执行后，其设置了在 `cpu_online_bits` cpumask 中的 0 位。这意味着第一个 cpu 此时上线了。

当然，除了 `set_cpu_*` API 外，cpumask 提供了其它 cpumasks 操作的 API。让我们简短看下。

附加的 cpumask API

cpumaks 提供了一系列宏来得到不同状态 CPUs 序号。例如：

```
#define num_online_cpus()    cpumask_weight(cpu_online_mask)
```

这个宏返回了 `online` CPUs 数量。它读取 `cpu_online_mask` 位图并调用了 `cpumask_weight` 函数。`cpumask_weight` 函数使用两个参数调用了一次 `bitmap_weight` 函数：

- cpumask bitmap;
- `nr_cpumask_bits` - 在我们的例子中就是 `NR_CPUS` 。

```
static inline unsigned int cpumask_weight(const struct cpumask *srcp)
{
    return bitmap_weight(cpumask_bits(srcp), nr_cpumask_bits);
}
```

并计算给定位图的位数。除了 `num_online_cpus` ，cpumask还提供了所有 CPU 状态的宏：

- num_possible_cpus;
- num_active_cpus;
- cpu_online;
- cpu_possible.

等等。

除了 Linux 内核提供的下述操作 `cpumask` 的 API：

- `for_each_cpu` - 遍历一个mask的所有 cpu;
- `for_each_cpu_not` - 遍历所有补集的 cpu;
- `cpumask_clear_cpu` - 清除一个 cpumask 的 cpu;
- `cpumask_test_cpu` - 测试一个 mask 中的 cpu;
- `cpumask_setall` - 设置 mask 的所有 cpu;
- `cpumask_size` - 返回分配 'struct cpumask' 字节数大小;

还有很多。

链接

- cpumask documentation

initcall 机制

介绍

就像你从标题所理解的，这部分将涉及 Linux 内核中有趣且重要的概念，称之为 `initcall` 。在 Linux 内核中，我们可以看到类似这样的定义：

```
early_param("debug", debug_kernel);
```

或者

```
arch_initcall(init_pit_clocksource);
```

在我们分析这个机制在内核中是如何实现的之前，我们必须了解这个机制是什么，以及在 Linux 内核中是如何使用它的。像这样的定义表示一个 回调函数 ，它们会在 Linux 内核启动中或启动后调用。实际上 `initcall` 机制的要点是确定内置模块和子系统初始化的正确顺序。举个例子，我们来看看下面的函数：

```
static int __init nmi_warning_debugfs(void)
{
    debugfs_create_u64("nmi_longest_ns", 0644,
                        arch_debugfs_dir, &nmi_longest_ns);
    return 0;
}
```

这个函数出自源码文件 arch/x86/kernel/nmi.c。我们可以看到，这个函数只是在 `arch_debugfs_dir` 目录中创建 `nmi_longest_ns` debugfs 文件。实际上，只有在 `arch_debugfs_dir` 创建后，才会创建这个 `debugfs` 文件。这个目录是在 Linux 内核特定架构的初始化期间创建的。实际上，该目录将在源码文件 arch/x86/kernel/kdebugfs.c 的 `arch_kdebugfs_init` 函数中创建。注意 `arch_kdebugfs_init` 函数也被标记为 `initcall` 。

```
arch_initcall(arch_kdebugfs_init);
```

Linux 内核在调用 `fs` 相关的 `initcalls` 之前调用所有特定架构的 `initcalls` 。因此，只有在 `arch_kdebugfs_dir` 目录创建以后才会创建我们的 `nmi_longest_ns` 。实际上，Linux 内核提供了八个级别的主 `initcalls` ：

- `early` ；
- `core` ；

- `postcore`；
- `arch`；
- `susys`；
- `fs`；
- `device`；
- `late`.

它们的所有名称是由数组 `initcall_level_names` 来描述的，该数组定义在源码文件 init/main.c 中：

```
static char *initcall_level_names[] __initdata = {
    "early",
    "core",
    "postcore",
    "arch",
    "subsys",
    "fs",
    "device",
    "late",
};
```

所有用这些标识符标记为 `initcall` 的函数将会以相同的顺序被调用，或者说，`early initcalls` 会首先被调用，其次是 `core initcalls`，以此类推。现在，我们对 `initcall` 机制了解点了，所以我们可以开始潜入 Linux 内核源码，来看看这个机制是如何实现的。

initcall 机制在 Linux 内核中的实现

Linux 内核提供了一组来自头文件 include/linux/init.h 的宏，来标记给定的函数为 `initcall`。所有这些宏都相当简单：

```
#define early_initcall(fn)        __define_initcall(fn, early)
#define core_initcall(fn)        __define_initcall(fn, 1)
#define postcore_initcall(fn)        __define_initcall(fn, 2)
#define arch_initcall(fn)        __define_initcall(fn, 3)
#define subsys_initcall(fn)        __define_initcall(fn, 4)
#define fs_initcall(fn)        __define_initcall(fn, 5)
#define device_initcall(fn)        __define_initcall(fn, 6)
#define late_initcall(fn)        __define_initcall(fn, 7)
```

我们可以看到，这些宏只是从同一个头文件的 `__define_initcall` 宏的调用扩展而来。此外，`__define_initcall` 宏有两个参数：

- `fn` - 在调用某个级别 `initcalls` 时调用的回调函数；
- `id` - 识别 `initcall` 的标识符，用来防止两个相同的 `initcalls` 指向同一个处理函数

时出现错误。

`__define_initcall` 宏的实现如下所示：

```
#define __define_initcall(fn, id) \
    static initcall_t __initcall_##fn##id __used \
    __attribute__((__section__(".initcall" #id ".init"))) = fn; \
    LTO_REFERENCE_INITCALL(__initcall_##fn##id)
```

要了解 `__define_initcall` 宏，首先让我们来看下 `initcall_t` 类型。这个类型定义在同一个 头文件 中，它表示一个返回 整形指针的函数指针，这将是 `initcall` 的结果：

```
typedef int (*initcall_t)(void);
```

现在让我们回到 `_-define_initcall` 宏。## 提供了连接两个符号的能力。在我们的例子中，`__define_initcall` 宏的第一行产生了 `.initcall id .init` ELF 部分 给定函数的定义，并标记以下 gcc 属性： `__initcall_function_name_id` 和 `__used` 。如果我们查看表示内核链接脚本数据的 include/asm-generic/vmlinux.lds.h 头文件，我们会看到所有的 `initcalls` 部分都将放在 `.data` 段：

```
#define INIT_CALLS                          \
        VMLINUX_SYMBOL(__initcall_start) = .;     \
        *(.initcallearly.init)                     \
        INIT_CALLS_LEVEL(0)                         \
        INIT_CALLS_LEVEL(1)                         \
        INIT_CALLS_LEVEL(2)                         \
        INIT_CALLS_LEVEL(3)                         \
        INIT_CALLS_LEVEL(4)                         \
        INIT_CALLS_LEVEL(5)                         \
        INIT_CALLS_LEVEL(rootfs)                 \
        INIT_CALLS_LEVEL(6)                         \
        INIT_CALLS_LEVEL(7)                         \
        VMLINUX_SYMBOL(__initcall_end) = .;

#define INIT_DATA_SECTION(initsetup_align)    \
    .init.data : AT(ADDR(.init.data) - LOAD_OFFSET) {       \
        ...                                               \
        INIT_CALLS                                          \
        ...                                               \
    }
```

第二个属性 - `__used` ，定义在 include/linux/compiler-gcc.h 头文件中，它扩展了以下 `gcc` 定义：

```
#define __used   __attribute__((__used__))
```

它防止 `定义了变量但未使用` 的告警。宏 `__define_initcall` 最后一行是：

```
LTO_REFERENCE_INITCALL(__initcall_##fn##id)
```

这取决于 `CONFIG_LTO` 内核配置选项，只为编译器提供链接时间优化存根：

```
#ifdef CONFIG_LTO
#define LTO_REFERENCE_INITCALL(x) \
        static __used __exit void *reference_##x(void)  \
        {                                                \
                return &x;                               \
        }
#else
#define LTO_REFERENCE_INITCALL(x)
#endif
```

为了防止当模块中的变量没有引用时而产生的任何问题，它被移到了程序末尾。这就是关于 `__define_initcall` 宏的全部了。所以，所有的 `*_initcall` 宏将会在Linux内核编译时扩展，所有的 `initcalls` 会放置在它们的段内，并可以通过 `.data` 段来获取，Linux 内核在初始化过程中就知道在哪儿去找到 `initcall` 并调用它。

既然 Linux 内核可以调用 `initcalls`，我们就来看下 Linux 内核是如何做的。这个过程从 init/main.c 头文件的 `do_basic_setup` 函数开始：

```
static void __init do_basic_setup(void)
{
    ...
    ...
    ...
       do_initcalls();
    ...
    ...
    ...
}
```

该函数在 Linux 内核初始化过程中调用，调用时机是主要的初始化步骤，比如内存管理器相关的初始化、`CPU` 子系统等完成之后。`do_initcalls` 函数只是遍历 `initcall` 级别数组，并调用每个级别的 `do_initcall_level` 函数：

```
static void __init do_initcalls(void)
{
    int level;

    for (level = 0; level < ARRAY_SIZE(initcall_levels) - 1; level++)
        do_initcall_level(level);
}
```

`initcall_levels` 数组在同一个源码文件中定义，包含了定义在 `__define_initcall` 宏中的那些段的指针：

```
static initcall_t *initcall_levels[] __initdata = {
    __initcall0_start,
    __initcall1_start,
    __initcall2_start,
    __initcall3_start,
    __initcall4_start,
    __initcall5_start,
    __initcall6_start,
    __initcall7_start,
    __initcall_end,
};
```

如果你有兴趣，你可以在 Linux 内核编译后生成的链接器脚本 `arch/x86/kernel/vmlinux.lds` 中找到这些段：

```
.init.data : AT(ADDR(.init.data) - 0xffffffff80000000) {
    ...
    ...
    ...
    ...
    __initcall_start = .;
    *(.initcallearly.init)
    __initcall0_start = .;
    *(.initcall0.init)
    *(.initcall0s.init)
    __initcall1_start = .;
    ...
    ...
}
```

如果你对这些不熟，可以在本书的某些部分了解更多关于链接器的信息。

正如我们刚看到的， `do_initcall_level` 函数有一个参数 - `initcall` 的级别，做了以下两件事：首先这个函数拷贝了 `initcall_command_line` ，这是通常内核包含了各个模块参数的命令行的副本，并用 kernel/params.c源码文件的 `parse_args` 函数解析它，然后调用各个级别的 `do_on_initcall` 函数：

```
for (fn = initcall_levels[level]; fn < initcall_levels[level+1]; fn++)
        do_one_initcall(*fn);
```

`do_on_initcall` 为我们做了主要的工作。我们可以看到，这个函数有一个参数表示 `initcall` 回调函数，并调用给定的回调函数：

```
int __init_or_module do_one_initcall(initcall_t fn)
{
    int count = preempt_count();
    int ret;
    char msgbuf[64];

    if (initcall_blacklisted(fn))
        return -EPERM;

    if (initcall_debug)
        ret = do_one_initcall_debug(fn);
    else
        ret = fn();

    msgbuf[0] = 0;

    if (preempt_count() != count) {
        sprintf(msgbuf, "preemption imbalance ");
        preempt_count_set(count);
    }
    if (irqs_disabled()) {
        strlcat(msgbuf, "disabled interrupts ", sizeof(msgbuf));
        local_irq_enable();
    }
    WARN(msgbuf[0], "initcall %pF returned with %s\n", fn, msgbuf);

    return ret;
}
```

让我们来试着理解 `do_on_initcall` 函数做了什么。首先我们增加 preemption 计数，以便我们稍后进行检查，确保它不是不平衡的。这步以后，我们可以看到 `initcall_backlist` 函数的调用，这个函数遍历包含了 `initcalls` 黑名单的 `blacklisted_initcalls` 链表，如果 `initcall` 在黑名单里就释放它：

```
list_for_each_entry(entry, &blacklisted_initcalls, next) {
    if (!strcmp(fn_name, entry->buf)) {
        pr_debug("initcall %s blacklisted\n", fn_name);
        kfree(fn_name);
        return true;
    }
}
```

黑名单的 `initcalls` 保存在 `blacklisted_initcalls` 链表中，这个链表是在早期 Linux 内核初始化时由 Linux 内核命令行来填充的。

处理完进入黑名单的 `initcalls` ，接下来的代码直接调用 `initcall` ：

```
if (initcall_debug)
    ret = do_one_initcall_debug(fn);
else
    ret = fn();
```

取决于 `initcall_debug` 变量的值， `do_one_initcall_debug` 函数将调用 `initcall` ，或直接调用 `fn()` 。 `initcall_debug` 变量定义在同一个源码文件：

```
bool initcall_debug;
```

该变量提供了向内核日志缓冲区打印一些信息的能力。可以通过 `initcall_debug` 参数从内核命令行中设置这个变量的值。从Linux内核命令行文档可以看到：

```
initcall_debug    [KNL] Trace initcalls as they are executed.  Useful
                      for working out where the kernel is dying during
                      startup.
```

确实如此。如果我们看下 `do_one_initcall_debug` 函数的实现，我们会看到它与 `do_one_initcall` 函数做了一样的事，也就是说， `do_one_initcall_debug` 函数调用了给定的 `initcall` ，并打印了一些和 `initcall` 相关的信息（比如当前任务的 pid 、 `initcall` 的持续时间等）：

```
static int __init_or_module do_one_initcall_debug(initcall_t fn)
{
    ktime_t calltime, delta, rettime;
    unsigned long long duration;
    int ret;

    printk(KERN_DEBUG "calling  %pF @ %i\n", fn, task_pid_nr(current));
    calltime = ktime_get();
    ret = fn();
    rettime = ktime_get();
    delta = ktime_sub(rettime, calltime);
    duration = (unsigned long long) ktime_to_ns(delta) >> 10;
    printk(KERN_DEBUG "initcall %pF returned %d after %lld usecs\n",
         fn, ret, duration);

    return ret;
}
```

由于 `initcall` 被 `do_one_initcall` 或 `do_one_initcall_debug` 调用，我们可以看到在 `do_one_initcall` 函数末尾做了两次检查。第一个检查在initcall执行内部 `__preempt_count_add` 和 `__preempt_count_sub` 可能的执行次数，如果这个值和之前的可抢占计数不相等，我们就把 `preemption imbalance` 字符串添加到消息缓冲区，并设置正确的可抢占计数：

```
if (preempt_count() != count) {
    sprintf(msgbuf, "preemption imbalance ");
    preempt_count_set(count);
}
```

稍后这个错误字符串就会被打印出来。最后检查本地 IRQs 的状态，如果它们被禁用了，我们就将 `disabled interrupts` 字符串添加到我们的消息缓冲区，并为当前处理器使能 `IRQs`，以防出现 `IRQs` 被 `initcall` 禁用了但不再使能的情况出现：

```
if (irqs_disabled()) {
    strlcat(msgbuf, "disabled interrupts ", sizeof(msgbuf));
    local_irq_enable();
}
```

这就是全部了。通过这种方式，Linux 内核以正确的顺序完成了很多子系统的初始化。现在我们知道 Linux 内核的 `initcall` 机制是怎么回事了。在这部分中，我们介绍了 `initcall` 机制的主要部分，但遗留了一些重要的概念。让我们来简单看下这些概念。

首先，我们错过了一个级别的 `initcalls`，就是 `rootfs initcalls`。和我们在本部分看到的很多宏类似，你可以在 include/linux/init.h 头文件中找到 `rootfs_initcall` 的定义：

```
#define rootfs_initcall(fn)        __define_initcall(fn, rootfs)
```

从这个宏的名字我们可以理解到，它的主要目的是保存和 rootfs 相关的回调。除此之外，只有在与设备相关的东西没被初始化时，在文件系统级别初始化以后再初始化一些其它东西时才有用。例如，发生在源码文件 init/initramfs.c 中 `populate_rootfs` 函数里的解压 initramfs：

```
rootfs_initcall(populate_rootfs);
```

在这里，我们可以看到熟悉的输出：

```
[    0.199960] Unpacking initramfs...
```

除了 `rootfs_initcall` 级别，还有其它的 `console_initcall` 、 `security_initcall` 和其他辅助的 `initcall` 级别。我们遗漏的最后一件事，是 `*_initcall_sync` 级别的集合。在这部分我们看到的几乎每个 `*_initcall` 宏，都有 `_sync` 前缀的宏伴随：

```
#define core_initcall_sync(fn)        __define_initcall(fn, 1s)
#define postcore_initcall_sync(fn)    __define_initcall(fn, 2s)
#define arch_initcall_sync(fn)        __define_initcall(fn, 3s)
#define subsys_initcall_sync(fn)    __define_initcall(fn, 4s)
#define fs_initcall_sync(fn)        __define_initcall(fn, 5s)
#define device_initcall_sync(fn)    __define_initcall(fn, 6s)
#define late_initcall_sync(fn)        __define_initcall(fn, 7s)
```

这些附加级别的主要目的是，等待所有某个级别的与模块相关的初始化例程完成。

这就是全部了。

结论

在这部分中，我们看到了 Linux 内核的一项重要机制，即在初始化期间允许调用依赖于 Linux 内核当前状态的函数。

如果你有问题或建议，可随时在 twitter 0xAX 上联系我，给我发 email，或者创建 issue。

请注意英语不是我的母语，对此带来的不便，我很抱歉。如果你发现了任何错误，都可以给我发 **PR** 到**linux-insides**。.

链接

- callback
- debugfs
- integer type
- symbols concatenation
- GCC
- Link time optimization
- Introduction to linkers
- Linux kernel command line
- Process identifier
- IRQs
- rootfs
- previous part

Notification Chains in Linux Kernel

Introduction

The Linux kernel is huge piece of C) code which consists from many different subsystems. Each subsystem has its own purpose which is independent of other subsystems. But often one subsystem wants to know something from other subsystem(s). There is special mechanism in the Linux kernel which allows to solve this problem partly. The name of this mechanism is - `notification chains` and its main purpose to provide a way for different subsystems to subscribe on asynchronous events from other subsystems. Note that this mechanism is only for communication inside kernel, but there are other mechanisms for communication between kernel and userspace.

Before we will consider `notification chains` API and implementation of this API, let's look at `Notification chains` mechanism from theoretical side as we did it in other parts of this book. Everything which is related to `notification chains` mechanism is located in the include/linux/notifier.h header file and kernel/notifier.c source code file. So let's open them and start to dive.

Notification Chains related data structures

Let's start to consider `notification chains` mechanism from related data structures. As I wrote above, main data structures should be located in the include/linux/notifier.h header file, so the Linux kernel provides generic API which does not depend on certain architecture. In general, the `notification chains` mechanism represents a list (that's why it named `chains`) of callback) functions which are will be executed when an event will be occurred.

All of these callback functions are represented as `notifier_fn_t` type in the Linux kernel:

```
typedef    int (*notifier_fn_t)(struct notifier_block *nb, unsigned long action, void
*data);
```

So we may see that it takes three following arguments:

- `nb` - is linked list of function pointers (will see it now);
- `action` - is type of an event. A notification chain may support multiple events, so we need this parameter to distinguish an event from other events;
- `data` - is storage for private information. Actually it allows to provide additional data

information about an event.

Additionally we may see that `notifier_fn_t` returns an integer value. This integer value maybe one of:

* `NOTIFY_DONE` - subscriber does not interested in notification;
* `NOTIFY_OK` - notification was processed correctly;
* `NOTIFY_BAD` - something went wrong;
* `NOTIFY_STOP` - notification is done, but no further callbacks should be called for this event.

All of these results defined as macros in the include/linux/notifier.h header file:

```
#define NOTIFY_DONE        0x0000
#define NOTIFY_OK        0x0001
#define NOTIFY_BAD        (NOTIFY_STOP_MASK|0x0002)
#define NOTIFY_STOP        (NOTIFY_OK|NOTIFY_STOP_MASK)
```

Where `NOTIFY_STOP_MASK` represented by the:

```
#define NOTIFY_STOP_MASK    0x8000
```

macro and means that callbacks will not be called during next notifications.

Each part of the Linux kernel which wants to be notified on a certain event will should provide own `notifier_fn_t` callback function. Main role of the `notification chains` mechanism is to call certain callbacks when an asynchronous event occurred.

The main building block of the `notification chains` mechanism is the `notifier_block` structure:

```
struct notifier_block {
    notifier_fn_t notifier_call;
    struct notifier_block __rcu *next;
    int priority;
};
```

which is defined in the include/linux/notifier.h file. This struct contains pointer to callback function - `notifier_call`, link to the next notification callback and `priority` of a callback function as functions with higher priority are executed first.

The Linux kernel provides notification chains of four following types:

* Blocking notifier chains;
* SRCU notifier chains;

- Atomic notifier chains;
- Raw notifier chains.

Let's consider all of these types of notification chains by order:

In the first case for the `blocking notifier chains`, callbacks will be called/executed in process context. This means that the calls in a notification chain may be blocked.

The second `SRCU notifier chains` represent alternative form of `blocking notifier chains`. In the first case, blocking notifier chains uses `rw_semaphore` synchronization primitive to protect chain links. `SRCU` notifier chains run in process context too, but uses special form of RCU mechanism which is permissible to block in an read-side critical section.

In the third case for the `atomic notifier chains` runs in interrupt or atomic context and protected by spinlock synchronization primitive. The last `raw notifier chains` provides special type of notifier chains without any locking restrictions on callbacks. This means that protection rests on the shoulders of caller side. It is very useful when we want to protect our chain with very specific locking mechanism.

If we will look at the implementation of the `notifier_block` structure, we will see that it contains pointer to the `next` element from a notification chain list, but we have no head. Actually a head of such list is in separate structure depends on type of a notification chain. For example for the `blocking notifier chains`:

```
struct blocking_notifier_head {
    struct rw_semaphore rwsem;
    struct notifier_block __rcu *head;
};
```

or for `atomic notification chains`:

```
struct atomic_notifier_head {
    spinlock_t lock;
    struct notifier_block __rcu *head;
};
```

Now as we know a little about `notification chains` mechanism let's consider implementation of its API.

Notification Chains

Usually there are two sides in a publish/subscriber mechanisms. One side who wants to get notifications and other side(s) who generates these notifications. We will consider notification chains mechanism from both sides. We will consider `blocking notification chains` in this part, because of other types of notification chains are similar to it and differs mostly in protection mechanisms.

Before a notification producer is able to produce notification, first of all it should initialize head of a notification chain. For example let's consider notification chains related to kernel loadable modules. If we will look in the kernel/module.c source code file, we will see following definition:

```
static BLOCKING_NOTIFIER_HEAD(module_notify_list);
```

which defines head for loadable modules blocking notifier chain. The `BLOCKING_NOTIFIER_HEAD` macro is defined in the include/linux/notifier.h header file and expands to the following code:

```
#define BLOCKING_INIT_NOTIFIER_HEAD(name) do {    \
        init_rwsem(&(name)->rwsem);                              \
        (name)->head = NULL;                                 \
    } while (0)
```

So we may see that it takes name of a name of a head of a blocking notifier chain and initializes read/write semaphore and set head to `NULL`. Besides the `BLOCKING_INIT_NOTIFIER_HEAD` macro, the Linux kernel additionally provides `ATOMIC_INIT_NOTIFIER_HEAD`, `RAW_INIT_NOTIFIER_HEAD` macros and `srcu_init_notifier` function for initialization atomic and other types of notification chains.

After initialization of a head of a notification chain, a subsystem which wants to receive notification from the given notification chain it should register with certain function which is depends on type of notification. If you will look in the include/linux/notifier.h header file, you will see following four function for this:

```
extern int atomic_notifier_chain_register(struct atomic_notifier_head *nh,
        struct notifier_block *nb);

extern int blocking_notifier_chain_register(struct blocking_notifier_head *nh,
        struct notifier_block *nb);

extern int raw_notifier_chain_register(struct raw_notifier_head *nh,
        struct notifier_block *nb);

extern int srcu_notifier_chain_register(struct srcu_notifier_head *nh,
        struct notifier_block *nb);
```

As I already wrote above, we will cover only blocking notification chains in the part, so let's consider implementation of the `blocking_notifier_chain_register` function. Implementation of this function is located in the kernel/notifier.c source code file and as we may see the `blocking_notifier_chain_register` takes two parameters:

- `nh` - head of a notification chain;
- `nb` - notification descriptor.

Now let's look at the implementation of the `blocking_notifier_chain_register` function:

```
int raw_notifier_chain_register(struct raw_notifier_head *nh,
        struct notifier_block *n)
{
    return notifier_chain_register(&nh->head, n);
}
```

As we may see it just returns result of the `notifier_chain_register` function from the same source code file and as we may understand this function does all job for us. Definition of the `notifier_chain_register` function looks:

```
int blocking_notifier_chain_register(struct blocking_notifier_head *nh,
        struct notifier_block *n)
{
    int ret;

    if (unlikely(system_state == SYSTEM_BOOTING))
        return notifier_chain_register(&nh->head, n);

    down_write(&nh->rwsem);
    ret = notifier_chain_register(&nh->head, n);
    up_write(&nh->rwsem);
    return ret;
}
```

As we may see implementation of the `blocking_notifier_chain_register` is pretty simple. First of all there is check which check current system state and if a system in rebooting state we just call the `notifier_chain_register`. In other way we do the same call of the `notifier_chain_register` but as you may see this call is protected with read/write semaphores. Now let's look at the implementation of the `notifier_chain_register` function:

```
static int notifier_chain_register(struct notifier_block **nl,
        struct notifier_block *n)
{
    while ((*nl) != NULL) {
        if (n->priority > (*nl)->priority)
            break;
        nl = &((*nl)->next);
    }
    n->next = *nl;
    rcu_assign_pointer(*nl, n);
    return 0;
}
```

This function just inserts new `notifier_block` (given by a subsystem which wants to get notifications) to the notification chain list. Besides subscribing on an event, subscriber may unsubscribe from a certain events with the set of `unsubscribe` functions:

```
extern int atomic_notifier_chain_unregister(struct atomic_notifier_head *nh,
        struct notifier_block *nb);

extern int blocking_notifier_chain_unregister(struct blocking_notifier_head *nh,
        struct notifier_block *nb);

extern int raw_notifier_chain_unregister(struct raw_notifier_head *nh,
        struct notifier_block *nb);

extern int srcu_notifier_chain_unregister(struct srcu_notifier_head *nh,
        struct notifier_block *nb);
```

When a producer of notifications wants to notify subscribers about an event, the `*.notifier_call_chain` function will be called. As you already may guess each type of notification chains provides own function to produce notification:

```
extern int atomic_notifier_call_chain(struct atomic_notifier_head *nh,
        unsigned long val, void *v);

extern int blocking_notifier_call_chain(struct blocking_notifier_head *nh,
        unsigned long val, void *v);

extern int raw_notifier_call_chain(struct raw_notifier_head *nh,
        unsigned long val, void *v);

extern int srcu_notifier_call_chain(struct srcu_notifier_head *nh,
        unsigned long val, void *v);
```

Let's consider implementation of the `blocking_notifier_call_chain` function. This function is defined in the kernel/notifier.c source code file:

```
int blocking_notifier_call_chain(struct blocking_notifier_head *nh,
        unsigned long val, void *v)
{
    return __blocking_notifier_call_chain(nh, val, v, -1, NULL);
}
```

and as we may see it just returns result of the `__blocking_notifier_call_chain` function. As we may see, the `blocking_notifer_call_chain` takes three parameters:

- `nh` - head of notification chain list;
- `val` - type of a notification;
- `v` - input parameter which may be used by handlers.

But the `__blocking_notifier_call_chain` function takes five parameters:

```
int __blocking_notifier_call_chain(struct blocking_notifier_head *nh,
                   unsigned long val, void *v,
                   int nr_to_call, int *nr_calls)
{
    ...
    ...
    ...
}
```

Where `nr_to_call` and `nr_calls` are number of notifier functions to be called and number of sent notifications. As you may guess the main goal of the `__blocking_notifer_call_chain` function and other functions for other notification types is to call callback function when an event occurred. Implementation of the `__blocking_notifier_call_chain` is pretty simple, it just calls the `notifier_call_chain` function from the same source code file protected with read/write semaphore:

```
int __blocking_notifier_call_chain(struct blocking_notifier_head *nh,
                   unsigned long val, void *v,
                   int nr_to_call, int *nr_calls)
{
    int ret = NOTIFY_DONE;

    if (rcu_access_pointer(nh->head)) {
        down_read(&nh->rwsem);
        ret = notifier_call_chain(&nh->head, val, v, nr_to_call,
                    nr_calls);
        up_read(&nh->rwsem);
    }
    return ret;
}
```

and returns its result. In this case all job is done by the `notifier_call_chain` function. Main purpose of this function informs registered notifiers about an asynchronous event:

```
static int notifier_call_chain(struct notifier_block **nl,
                    unsigned long val, void *v,
                    int nr_to_call, int *nr_calls)
{
    ...
    ...
    ...
    ret = nb->notifier_call(nb, val, v);
    ...
    ...
    ...
    return ret;
}
```

That's all. In generall all looks pretty simple.

Now let's consider on a simple example related to loadable modules. If we will look in the kernel/module.c. As we already saw in this part, there is:

```
static BLOCKING_NOTIFIER_HEAD(module_notify_list);
```

definition of the `module_notify_list` in the kernel/module.c source code file. This definition determines head of list of blocking notifier chains related to kernel modules. There are at least three following events:

- MODULE_STATE_LIVE
- MODULE_STATE_COMING
- MODULE_STATE_GOING

in which maybe interested some subsystems of the Linux kernel. For example tracing of kernel modules states. Instead of direct call of the `atomic_notifier_chain_register`, `blocking_notifier_chain_register` and etc., most notification chains come with a set of wrappers used to register to them. Registatrion on these modules events is going with the help of such wrapper:

```
int register_module_notifier(struct notifier_block *nb)
{
    return blocking_notifier_chain_register(&module_notify_list, nb);
}
```

If we will look in the kernel/tracepoint.c source code file, we will see such registration during initialization of tracepoints:

```
static __init int init_tracepoints(void)
{
    int ret;

    ret = register_module_notifier(&tracepoint_module_nb);
    if (ret)
        pr_warn("Failed to register tracepoint module enter notifier\n");

    return ret;
}
```

Where `tracepoint_module_nb` provides callback function:

```
static struct notifier_block tracepoint_module_nb = {
    .notifier_call = tracepoint_module_notify,
    .priority = 0,
};
```

When one of the `MODULE_STATE_LIVE` , `MODULE_STATE_COMING` or `MODULE_STATE_GOING` events occurred. For example the `MODULE_STATE_LIVE` the `MODULE_STATE_COMING` notifications will be sent during execution of the init_module system call. Or for example `MODULE_STATE_GOING` will be sent during execution of the delete_module `system call` :

```
SYSCALL_DEFINE2(delete_module, const char __user *, name_user,
        unsigned int, flags)
{
    ...
    ...
    ...
    blocking_notifier_call_chain(&module_notify_list,
                 MODULE_STATE_GOING, mod);
    ...
    ...
    ...
}
```

Thus when one of these system call will be called from userspace, the Linux kernel will send certain notification depends on a system call and the `tracepoint_module_notify` callback function will be called.

That's all.

Links

- C programming langauge)

- API
- callback)
- RCU
- spinlock
- loadable modules
- semaphore
- tracepoints
- system call
- init_module system call
- delete_module
- previous part

Linux内核中的数据结构

Linux内核对很多数据结构提供不同的实现方法，比如，双向链表，B+树，具有优先级的堆等等。

这部分考虑这些数据结构和算法。

- 双向链表
- 基数树
- 位数组

Linux 内核里的数据结构——双向链表

双向链表

Linux 内核自己实现了双向链表，可以在 include/linux/list.h 找到定义。我们将会从双向链表数据结构开始 `内核的数据结构` 。为什么？因为它在内核里使用的很广泛，你只需要在 free-electrons.com 检索一下就知道了。

首先让我们看一下在 include/linux/types.h 里的主结构体：

```
struct list_head {
    struct list_head *next, *prev;
};
```

你可能注意到这和你以前见过的双向链表的实现方法是不同的。举个例子来说，在 glib 库里是这样实现的：

```
struct GList {
  gpointer data;
  GList *next;
  GList *prev;
};
```

通常来说一个链表会包含一个指向某个项目的指针。但是内核的实现并没有这样做。所以问题来了： `链表在哪里保存数据呢？` 。实际上内核里实现的链表实际上是 `侵入式链表` 。侵入式链表并不在节点内保存数据-节点仅仅包含指向前后节点的指针，然后把数据是附加到链表的。这就使得这个数据结构是通用的，使用起来就不需要考虑节点数据的类型了。

比如：

```
struct nmi_desc {
    spinlock_t lock;
    struct list_head head;
};
```

让我们看几个例子来理解一下在内核里是如何使用 `list_head` 的。如上所述，在内核里有实在很多不同的地方用到了链表。我们以杂项字符驱动为例来说明双向链表的使用。在 drivers/char/misc.c 的杂项字符驱动API 被用来编写处理小型硬件和虚拟设备的小驱动。这些驱动共享相同的主设备号：

```
#define MISC_MAJOR              10
```

但是都有各自不同的次设备号。比如：

```
ls -l /dev |  grep 10
crw-------   1 root root     10, 235 Mar 21 12:01 autofs
drwxr-xr-x  10 root root         200 Mar 21 12:01 cpu
crw-------   1 root root     10,  62 Mar 21 12:01 cpu_dma_latency
crw-------   1 root root     10, 203 Mar 21 12:01 cuse
drwxr-xr-x   2 root root         100 Mar 21 12:01 dri
crw-rw-rw-   1 root root     10, 229 Mar 21 12:01 fuse
crw-------   1 root root     10, 228 Mar 21 12:01 hpet
crw-------   1 root root     10, 183 Mar 21 12:01 hwrng
crw-rw----+  1 root kvm      10, 232 Mar 21 12:01 kvm
crw-rw----   1 root disk     10, 237 Mar 21 12:01 loop-control
crw-------   1 root root     10, 227 Mar 21 12:01 mcelog
crw-------   1 root root     10,  59 Mar 21 12:01 memory_bandwidth
crw-------   1 root root     10,  61 Mar 21 12:01 network_latency
crw-------   1 root root     10,  60 Mar 21 12:01 network_throughput
crw-r-----   1 root kmem     10, 144 Mar 21 12:01 nvram
brw-rw----   1 root disk      1,  10 Mar 21 12:01 ram10
crw--w----   1 root tty       4,  10 Mar 21 12:01 tty10
crw-rw----   1 root dialout   4,  74 Mar 21 12:01 ttyS10
crw-------   1 root root     10,  63 Mar 21 12:01 vga_arbiter
crw-------   1 root root     10, 137 Mar 21 12:01 vhci
```

现在让我们看看它是如何使用链表的。首先看一下结构体 `miscdevice` ：

```
struct miscdevice
{
      int minor;
      const char *name;
      const struct file_operations *fops;
      struct list_head list;
      struct device *parent;
      struct device *this_device;
      const char *nodename;
      mode_t mode;
};
```

我们可以看到结构体的第四个变量 `list` 是所有注册过的设备的链表。在源代码文件的开始可以看到这个链表的定义：

```
static LIST_HEAD(misc_list);
```

它扩展开来实际上就是定义了一个 `list_head` 类型的变量：

```
#define LIST_HEAD(name) \
    struct list_head name = LIST_HEAD_INIT(name)
```

然后使用宏 `LIST_HEAD_INIT` 进行初始化，这会使用变量 `name` 的地址来填充 `prev` 和 `next` 结构体的两个变量。

```
#define LIST_HEAD_INIT(name) { &(name), &(name) }
```

现在来看看注册杂项设备的函数 `misc_register` 。它在开始就用 `INIT_LIST_HEAD` 初始化了 `miscdevice->list` 。

```
INIT_LIST_HEAD(&misc->list);
```

作用和宏 `LIST_HEAD_INIT` 一样。

```
static inline void INIT_LIST_HEAD(struct list_head *list)
{
    list->next = list;
    list->prev = list;
}
```

下一步在函数 `device_create` 创建了设备后我们就用下面的语句将设备添加到设备链表：

```
list_add(&misc->list, &misc_list);
```

内核文件 `list.h` 提供了向链表添加新项的接口函数。我们来看看它的实现：

```
static inline void list_add(struct list_head *new, struct list_head *head)
{
    __list_add(new, head, head->next);
}
```

实际上就是使用3个指定的参数来调用了内部函数 `__list_add` ：

- new - 新项。
- head - 新项将会被添加到 `head` 之后.
- head->next - `head` 之后的项。

`__list_add` 的实现非常简单：

```
static inline void __list_add(struct list_head *new,
                  struct list_head *prev,
                  struct list_head *next)
{
    next->prev = new;
    new->next = next;
    new->prev = prev;
    prev->next = new;
}
```

我们会在 `prev` 和 `next` 之间添加一个新项。所以我们用宏 `LIST_HEAD_INIT` 定义的 `misc` 链表会包含指向 `miscdevice->list` 的向前指针和向后指针。

这里仍有一个问题：如何得到列表的内容呢？这里有一个特殊的宏：

```
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)
```

使用了三个参数：

- ptr - 指向链表头的指针；
- type - 结构体类型;
- member - 在结构体内类型为 `list_head` 的变量的名字；

比如说：

```
const struct miscdevice *p = list_entry(v, struct miscdevice, list)
```

然后我们就可以使用 `p->minor` 或者 `p->name` 来访问 `miscdevice`。让我们来看看 `list_entry` 的实现：

```
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)
```

如我们所见，它仅仅使用相同的参数调用了宏 `container_of`。初看这个宏挺奇怪的：

```
#define container_of(ptr, type, member) ({                      \
    const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
    (type *)( (char *)__mptr - offsetof(type,member) );})
```

首先你可以注意到花括号内包含两个表达式。编译器会执行花括号内的全部语句，然后返回最后的表达式的值。

举个例子来说：

```
#include <stdio.h>

int main() {
    int i = 0;
    printf("i = %d\n", ({++i; ++i;}));
    return 0;
}
```

最终会打印 `2`

下一点就是 `typeof` ,它也很简单。就如你从名字所理解的，它仅仅返回了给定变量的类型。当我第一次看到宏 `container_of` 的实现时，让我觉得最奇怪的就是 `container_of` 中的 0 。实际上这个指针巧妙的计算了从结构体特定变量的偏移，这里的 `0` 刚好就是位宽里的零偏移。让我们看一个简单的例子：

```
#include <stdio.h>

struct s {
        int field1;
        char field2;
    char field3;
};

int main() {
    printf("%p\n", &((struct s*)0)->field3);
    return 0;
}
```

结果显示 `0x5` 。

下一个宏 `offsetof` 会计算从结构体的某个变量的相对于结构体起始地址的偏移。它的实现和上面类似：

```
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
```

现在我们来总结一下宏 `container_of` 。只需要知道结构体里面类型为 `list_head` 的变量的名字和结构体容器的类型，它可以通过结构体的变量 `list_head` 获得结构体的起始地址。在宏定义的第一行，声明了一个指向结构体成员变量 `ptr` 的指针 `__mptr` ，并且把 `ptr` 的地址赋给它。现在 `ptr` 和 `__mptr` 指向了同一个地址。从技术上讲我们并不需要这一行，但是它可以方便的进行类型检查。第一行保证了特定的结构体（参数 `type` ）包含成员变量 `member` 。第二行代码会用宏 `offsetof` 计算成员变量相对于结构体起始地址的偏移，然后从结构体的地址减去这个偏移，最后就得到了结构体的起始地址。

当然了 `list_add` 和 `list_entry` 不是 `<linux/list.h>` 提供的唯一函数。双向链表的实现还提供了如下API：

- list_add
- list_add_tail
- list_del
- list_replace
- list_move
- list_is_last
- list_empty
- list_cut_position
- list_splice
- list_for_each
- list_for_each_entry

等等很多其它 API。

Linux内核中的数据结构

基数树

正如你所知道的 Linux 内核通过许多不同库以及函数提供各种数据结构以及算法实现。 这个部分我们将介绍其中一个数据结构 Radix tree。Linux 内核中有两个文件与 `radix tree` 的实现和API相关：

- include/linux/radix-tree.h
- lib/radix-tree.c

首先说明一下什么是 `radix tree` 。Radix tree 是一种 `压缩 trie` ，其中 trie 是一种通过保存关联数组（associative array）来提供 `关键字-值（key-value）` 存储与查找的数据结构。通常关键字是字符串，不过也可以是其他数据类型。

trie 结构的节点与 `n-tree` 不同，其节点中并不存储关键字，取而代之的是存储单个字符标签。关键字查找时，通过从树的根开始遍历关键字相关的所有字符标签节点，直至到达最终的叶子节点。下面是个例子：

```
              +-----------+
              |           |
              |    " "    |
              |           |
       +------+-----------+------+
       |                         |
       |                         |
  +----v------+            +-----v-----+
  |           |            |           |
  |    g      |            |     c     |
  |           |            |           |
  +-----------+            +-----------+
       |                         |
       |                         |
  +----v------+            +-----v-----+
  |           |            |           |
  |    o      |            |     a     |
  |           |            |           |
  +-----------+            +-----------+
                                 |
                                 |
                           +-----v-----+
                           |           |
                           |     t     |
                           |           |
                           +-----------+
```

这个例子中，我们可以看到 `trie` 所存储的关键字信息 `go` 与 `cat`，压缩 trie 或 `radix tree` 与 `trie` 所不同的是，所有只存在单个孩子的中间节点将被压缩。

Linux 内核中的 Radix 树将值映射为整型关键字，Radix 的数据结构定义在 include/linux/radix-tree.h 文件中：

```
struct radix_tree_root {
         unsigned int            height;
         gfp_t                   gfp_mask;
         struct radix_tree_node  __rcu *rnode;
};
```

上面这个是 radix 树的 root 节点的结构体，它包括三个成员：

- `height` - 从叶节点向上计算出的树高度。
- `gfp_mask` - 内存分配标识。
- `rnode` - 子节点指针。

这里我们先讨论的结构体成员是 `gfp_mask`：

Linux 底层的内存申请接口需要提供一类标识（flag）- `gfp_mask` ，用于描述内存申请的行为。这个以 `GFP_` 前缀开头的内存申请控制标识主要包括，`GFP_NOIO` 禁止所有IO操作但允许睡眠等待内存，`__GFP_HIGHMEM` 允许申请内核的高端内存，`GFP_ATOMIC` 高优先级申请内存且操作不允许被睡眠。

接下来说的结构体成员是 `rnode` ：

```
struct radix_tree_node {
        unsigned int    path;
        unsigned int    count;
        union {
                struct {
                        struct radix_tree_node *parent;
                        void *private_data;
                };
                struct rcu_head rcu_head;
        };
        /* For tree user */
        struct list_head private_list;
        void __rcu      *slots[RADIX_TREE_MAP_SIZE];
        unsigned long   tags[RADIX_TREE_MAX_TAGS][RADIX_TREE_TAG_LONGS];
};
```

这个结构体中包括这几个内容，节点与父节点的偏移以及到树底端的高度，子节点的个数，节点的存储数据域，具体描述如下：

- `path` - 从叶节点
- `count` - 子节点的个数。
- `parent` - 父节点的指针。
- `private_data` - 存储数据内容缓冲区。
- `rcu_head` - 用于节点释放的RCU链表。
- `private_list` - 存储数据。

结构体 `radix_tree_node` 的最后两个成员 `tags` 与 `slots` 是非常重要且需要特别注意的。每个 Radix 树节点都可以包括一个指向存储数据指针的 slots 集合，空闲 slots 的指针指向 NULL。 Linux 内核的 Radix 树结构体中还包含用于记录节点存储状态的标签 `tags` 成员，标签通过位设置指示 Radix 树的数据存储状态。

至此，我们了解到 radix 树的结构，接下来看一下 radix 树所提供的 API。

Linux 内核基数树 API

我们从数据结构的初始化开始看，radix 树支持两种方式初始化。

第一个是使用宏 `RADIX_TREE` ：

```
RADIX_TREE(name, gfp_mask);
`
```

正如你看到，只需要提供 `name` 参数，就能够使用 `RADIX_TREE` 宏完成 radix 的定义以及初始化，`RADIX_TREE` 宏的实现非常简单：

```
#define RADIX_TREE(name, mask) \
         struct radix_tree_root name = RADIX_TREE_INIT(mask)

#define RADIX_TREE_INIT(mask)   { \
        .height = 0,              \
        .gfp_mask = (mask),       \
        .rnode = NULL,            \
}
```

`RADIX_TREE` 宏首先使用 `name` 定义了一个 `radix_tree_root` 实例并用 `RADIX_TREE_INIT` 宏带参数 `mask` 进行初始化。宏 `RADIX_TREE_INIT` 将 `radix_tree_root` 初始化为默认属性并将 gfp_mask 初始化为入参 `mask` 。第二种方式是手工定义 `radix_tree_root` 变量，之后再使用 `mask` 调用 `INIT_RADIX_TREE` 宏对变量进行初始化。

```
struct radix_tree_root my_radix_tree;
INIT_RADIX_TREE(my_tree, gfp_mask_for_my_radix_tree);
```

`INIT_RADIX_TREE` 宏定义：

```
#define INIT_RADIX_TREE(root, mask)  \
do {                                 \
        (root)->height = 0;          \
        (root)->gfp_mask = (mask);   \
        (root)->rnode = NULL;        \
} while (0)
```

宏 `INIT_RADIX_TREE` 所初始化的属性与 `RADIX_TREE_INIT` 一致

接下来是 radix 树的节点插入以及删除，这两个函数：

- `radix_tree_insert` ;
- `radix_tree_delete` .

第一个函数 `radix_tree_insert` 需要三个入参：

- radix 树 root 节点结构
- 索引关键字
- 需要插入存储的数据

第二个函数 `radix_tree_delete` 除了不需要存储数据参数外，其他与 `radix_tree_insert` 一致。

radix 树的查找实现有以下几个函数：The search in a radix tree implemented in two ways:

* `radix_tree_lookup`；
* `radix_tree_gang_lookup`；
* `radix_tree_lookup_slot`.

第一个函数 `radix_tree_lookup` 需要两个参数：

* radix 树 root 节点结构
* 索引关键字

这个函数通过给定的关键字查找 radix 树，并返关键字所对应的结点。

第二个函数 `radix_tree_gang_lookup` 具有以下特征：

```
unsigned int radix_tree_gang_lookup(struct radix_tree_root *root,
                                    void **results,
                                    unsigned long first_index,
                                    unsigned int max_items);
```

函数返回查找到记录的条目数，并根据关键字进行排序，返回的总结点数不超过入参 `max_items` 的大小。

最后一个函数 `radix_tree_lookup_slot` 返回结点 slot 中所存储的数据。

链接

* Radix tree
* Trie

Linux 内核里的数据结构——位数组

Linux 内核中的位数组和位操作

除了不同的基于链式和树的数据结构以外，Linux 内核也为位数组（或称为位图（bitmap））提供了 API。位数组在 Linux 内核里被广泛使用，并且在以下的源代码文件中包含了与这样的结构搭配使用的通用 `API`：

- lib/bitmap.c
- include/linux/bitmap.h

除了这两个文件之外，还有体系结构特定的头文件，它们为特定的体系结构提供优化的位操作。我们将探讨 x86_64 体系结构，因此在我们的例子里，它会是

- arch/x86/include/asm/bitops.h

头文件。正如我上面所写的，`位图` 在 Linux 内核中被广泛地使用。例如，`位数组` 常常用于保存一组在线/离线处理器，以便系统支持热插拔的 CPU（你可以在 cpumasks 部分阅读更多相关知识），一个位数组（bit array）可以在 Linux 内核初始化等期间保存一组已分配的中断处理。

因此，本部分的主要目的是了解位数组（bit array）是如何在 Linux 内核中实现的。让我们现在开始吧。

位数组声明

在我们开始查看 `位图` 操作的 `API` 之前，我们必须知道如何在 Linux 内核中声明它。有两种声明位数组的通用方法。第一种简单的声明一个位数组的方法是，定义一个 `unsigned long` 的数组，例如：

```
unsigned long my_bitmap[8]
```

第二种方法，是使用 `DECLARE_BITMAP` 宏，它定义于 include/linux/types.h 头文件：

```
#define DECLARE_BITMAP(name,bits) \
    unsigned long name[BITS_TO_LONGS(bits)]
```

我们可以看到 `DECLARE_BITMAP` 宏使用两个参数：

- `name` - 位图名称;
- `bits` - 位图中位数;

并且只是使用 `BITS_TO_LONGS(bits)` 元素展开 `unsigned long` 数组的定义。 `BITS_TO_LONGS` 宏将一个给定的位数转换为 `long` 的个数，换言之，就是计算 `bits` 中有多少个 `8` 字节元素：

```
#define BITS_PER_BYTE           8
#define DIV_ROUND_UP(n,d) (((n) + (d) - 1) / (d))
#define BITS_TO_LONGS(nr)       DIV_ROUND_UP(nr, BITS_PER_BYTE * sizeof(long))
```

因此，例如 `DECLARE_BITMAP(my_bitmap, 64)` 将产生：

```
>>> (((64) + (64) - 1) / (64))
1
```

与：

```
unsigned long my_bitmap[1];
```

在能够声明一个位数组之后，我们便可以使用它了。

体系结构特定的位操作

我们已经看了上面提及的一对源文件和头文件，它们提供了位数组操作的 API。其中重要且广泛使用的位数组 API 是体系结构特定的且位于已提及的头文件中 arch/x86/include/asm/bitops.h。

首先让我们查看两个最重要的函数：

- `set_bit`;
- `clear_bit`.

我认为没有必要解释这些函数的作用。从它们的名字来看，这已经很清楚了。让我们直接查看它们的实现。如果你浏览 arch/x86/include/asm/bitops.h 头文件，你将会注意到这些函数中的每一个都有原子性和非原子性两种变体。在我们开始深入这些函数的实现之前，首先，我们必须了解一些有关原子（atomic）操作的知识。

简而言之，原子操作保证两个或以上的操作不会并发地执行同一数据。 `x86` 体系结构提供了一系列原子指令，例如， xchg、cmpxchg 等指令。除了原子指令，一些非原子指令可以在 lock 指令的帮助下具有原子性。现在你已经对原子操作有了足够的了解，我们可以接着探讨 `set_bit` 和 `clear_bit` 函数的实现。

我们先考虑函数的非原子性（non-atomic）变体。非原子性的 `set_bit` 和 `clear_bit` 的名字以双下划线开始。正如我们所知道的，所有这些函数都定义于 arch/x86/include/asm/bitops.h 头文件，并且第一个函数就是 `__set_bit`：

```
static inline void __set_bit(long nr, volatile unsigned long *addr)
{
    asm volatile("bts %1,%0" : ADDR : "Ir" (nr) : "memory");
}
```

正如我们所看到的，它使用了两个参数：

- `nr` - 位数组中的位号（LCTT 译注：从 0开始）
- `addr` - 我们需要置位的位数组地址

注意，`addr` 参数使用 `volatile` 关键字定义，以告诉编译器给定地址指向的变量可能会被修改。 `__set_bit` 的实现相当简单。正如我们所看到的，它仅包含一行内联汇编代码。在我们的例子中，我们使用 bts 指令，从位数组中选出一个第一操作数（我们的例子中的 `nr`）所指定的位，存储选出的位的值到 CF 标志寄存器并设置该位（LCTT 译注：即 `nr` 指定的位置为 1）。

注意，我们了解了 `nr` 的用法，但这里还有一个参数 `addr` 呢！你或许已经猜到秘密就在 `ADDR`。 `ADDR` 是一个定义在同一个头文件中的宏，它展开为一个包含给定地址和 `+m` 约束的字符串：

```
#define ADDR                BITOP_ADDR(addr)
#define BITOP_ADDR(x) "+m" (*(volatile long *) (x))
```

除了 `+m` 之外，在 `__set_bit` 函数中我们可以看到其他约束。让我们查看并试着理解它们所表示的意义：

- `+m` - 表示内存操作数，这里的 `+` 表明给定的操作数为输入输出操作数；
- `I` - 表示整型常量；
- `r` - 表示寄存器操作数

除了这些约束之外，我们也能看到 `memory` 关键字，其告诉编译器这段代码会修改内存中的变量。到此为止，现在我们看看相同的原子性（atomic）变体函数。它看起来比非原子性（non-atomic）变体更加复杂：

```
static __always_inline void
set_bit(long nr, volatile unsigned long *addr)
{
    if (IS_IMMEDIATE(nr)) {
        asm volatile(LOCK_PREFIX "orb %1,%0"
            : CONST_MASK_ADDR(nr, addr)
            : "iq" ((u8)CONST_MASK(nr))
            : "memory");
    } else {
        asm volatile(LOCK_PREFIX "bts %1,%0"
            : BITOP_ADDR(addr) : "Ir" (nr) : "memory");
    }
}
```

（LCTT 译注：BITOP_ADDR 的定义为： `#define BITOP_ADDR(x) "=m" (*(volatile long *)(x))` ，ORB 为字节按位或。）

首先注意，这个函数使用了与 `__set_bit` 相同的参数集合，但额外地使用了 `__always_inline` 属性标记。 `__always_inline` 是一个定义于 include/linux/compiler-gcc.h 的宏，并且只是展开为 `always_inline` 属性：

```
#define __always_inline inline __attribute__((always_inline))
```

其意味着这个函数总是内联的，以减少 Linux 内核映像的大小。现在让我们试着了解下 `set_bit` 函数的实现。首先我们在 `set_bit` 函数的开头检查给定的位的数量。 `IS_IMMEDIATE` 宏定义于相同的头文件，并展开为 gcc 内置函数的调用：

```
#define IS_IMMEDIATE(nr)        (__builtin_constant_p(nr))
```

如果给定的参数是编译期已知的常量， `__builtin_constant_p` 内置函数则返回 `1` ，其他情况返回 `0` 。假若给定的位数是编译期已知的常量，我们便无须使用效率低下的 `bts` 指令去设置位。我们可以只需在给定地址指向的字节上执行 按位或 操作，其字节包含给定的位，掩码位数表示高位为 `1` ，其他位为 0 的掩码。在其他情况下，如果给定的位号不是编译期已知常量，我们便做和 `__set_bit` 函数一样的事。 `CONST_MASK_ADDR` 宏：

```
#define CONST_MASK_ADDR(nr, addr)    BITOP_ADDR((void *)(addr) + ((nr)>>3))
```

展开为带有到包含给定位的字节偏移的给定地址，例如，我们拥有地址 `0x1000` 和位号 `0x9` 。因为 `0x9` 代表 `一个字节 + 一位` ，所以我们的地址是 `addr + 1` :

```
>>> hex(0x1000 + (0x9 >> 3))
'0x1001'
```

`CONST_MASK` 宏将我们给定的位号表示为字节，位号对应位为高位 `1` ，其他位为 `0` ：

```
#define CONST_MASK(nr)            (1 << ((nr) & 7))
```

```
>>> bin(1 << (0x9 & 7))
'0b10'
```

最后，我们应用 `按位或` 运算到这些变量上面，因此，假如我们的地址是 `0x4097` ，并且我们需要置位号为 `9` 的位为 1：

```
>>> bin(0x4097)
'0b100000010010111'
>>> bin((0x4097 >> 0x9) | (1 << (0x9 & 7)))
'0b100010'
```

`第 9 位` 将会被置位。（LCTT 译注：这里的 9 是从 0 开始计数的，比如0010，按照作者的意思，其中的 1 是第 1 位）

注意，所有这些操作使用 `LOCK_PREFIX` 标记，其展开为 lock 指令，保证该操作的原子性。

正如我们所知，除了 `set_bit` 和 `__set_bit` 操作之外，Linux 内核还提供了两个功能相反的函数，在原子性和非原子性的上下文中清位。它们是 `clear_bit` 和 `__clear_bit` 。这两个函数都定义于同一个头文件 并且使用相同的参数集合。不仅参数相似，一般而言，这些函数与 `set_bit` 和 `__set_bit` 也非常相似。让我们查看非原子性 `__clear_bit` 的实现吧：

```
static inline void __clear_bit(long nr, volatile unsigned long *addr)
{
    asm volatile("btr %1,%0" : ADDR : "Ir" (nr));
}
```

没错，正如我们所见， `__clear_bit` 使用相同的参数集合，并包含极其相似的内联汇编代码块。它只是使用 btr 指令替换了 `bts` 。正如我们从函数名所理解的一样，通过给定地址，它清除了给定的位。 `btr` 指令表现得像 `bts` （LCTT 译注：原文这里为 btr，可能为笔误，修正为 bts）。该指令选出第一操作数所指定的位，存储它的值到 `CF` 标志寄存器，并且清除第二操作数指定的位数组中的对应位。

`__clear_bit` 的原子性变体为 `clear_bit` ：

```
static __always_inline void
clear_bit(long nr, volatile unsigned long *addr)
{
    if (IS_IMMEDIATE(nr)) {
        asm volatile(LOCK_PREFIX "andb %1,%0"
            : CONST_MASK_ADDR(nr, addr)
            : "iq" ((u8)~CONST_MASK(nr)));
    } else {
        asm volatile(LOCK_PREFIX "btr %1,%0"
            : BITOP_ADDR(addr)
            : "Ir" (nr));
    }
}
```

并且正如我们所看到的，它与 `set_bit` 非常相似，只有两处不同。第一处差异为 `clear_bit` 使用 `btr` 指令来清位，而 `set_bit` 使用 `bts` 指令来置位。第二处差异为 `clear_bit` 使用否定的位掩码和 `按位与` 在给定的字节上置位，而 `set_bit` 使用 `按位或` 指令。

到此为止，我们可以在任意位数组置位和清位了，我们将看看位掩码上的其他操作。

在 Linux 内核中对位数组最广泛使用的操作是设置和清除位，但是除了这两个操作外，位数组上其他操作也是非常有用的。Linux 内核里另一种广泛使用的操作是知晓位数组中一个给定的位是否被置位。我们能够通过 `test_bit` 宏的帮助实现这一功能。这个宏定义于 arch/x86/include/asm/bitops.h 头文件，并根据位号分别展开为 `constant_test_bit` 或 `variable_test_bit` 调用。

```
#define test_bit(nr, addr)              \
    (__builtin_constant_p((nr))                 \
     ? constant_test_bit((nr), (addr))            \
     : variable_test_bit((nr), (addr)))
```

因此，如果 `nr` 是编译期已知常量，`test_bit` 将展开为 `constant_test_bit` 函数的调用，而其他情况则为 `variable_test_bit`。现在让我们看看这些函数的实现，让我们从 `variable_test_bit` 开始看起：

```
static inline int variable_test_bit(long nr, volatile const unsigned long *addr)
{
    int oldbit;

    asm volatile("bt %2,%1\n\t"
             "sbb %0,%0"
             : "=r" (oldbit)
             : "m" (*(unsigned long *)addr), "Ir" (nr));

    return oldbit;
}
```

`variable_test_bit` 函数使用了与 `set_bit` 及其他函数使用的相似的参数集合。我们也可以看到执行 bt 和 sbb 指令的内联汇编代码。`bt` （或称 `bit test` ）指令从第二操作数指定的位数组选出第一操作数指定的一个指定位，并且将该位的值存进标志寄存器的 CF 位。第二个指令 `sbb` 从第二操作数中减去第一操作数，再减去 `CF` 的值。因此，这里将一个从给定位数组中的给定位号的值写进标志寄存器的 `CF` 位，并且执行 `sbb` 指令计算： `00000000 - CF` ，并将结果写进 `oldbit` 变量。

`constant_test_bit` 函数做了和我们在 `set_bit` 所看到的一样的事：

```
static __always_inline int constant_test_bit(long nr, const volatile unsigned long *addr)
{
    return ((1UL << (nr & (BITS_PER_LONG-1))) &
        (addr[nr >> _BITOPS_LONG_SHIFT])) != 0;
}
```

它生成了一个位号对应位为高位 `1` ，而其他位为 `0` 的字节（正如我们在 `CONST_MASK` 所看到的），并将 按位与 应用于包含给定位号的字节。

下一个被广泛使用的位数组相关操作是改变一个位数组中的位。为此，Linux 内核提供了两个辅助函数：

- `__change_bit` ；
- `change_bit` .

你可能已经猜测到，就拿 `set_bit` 和 `__set_bit` 例子说，这两个变体分别是原子和非原子版本。首先，让我们看看 `__change_bit` 函数的实现：

```
static inline void __change_bit(long nr, volatile unsigned long *addr)
{
    asm volatile("btc %1,%0" : ADDR : "Ir" (nr));
}
```

相当简单，不是吗？ `__change_bit` 的实现和 `__set_bit` 一样，只是我们使用 btc 替换 `bts` 指令而已。 该指令从一个给定位数组中选出一个给定位，将该为位的值存进 `CF` 并使用求反操作改变它的值，因此值为 `1` 的位将变为 `0` ，反之亦然：

```
>>> int(not 1)
0
>>> int(not 0)
1
```

`__change_bit` 的原子版本为 `change_bit` 函数：

```
static inline void change_bit(long nr, volatile unsigned long *addr)
{
    if (IS_IMMEDIATE(nr)) {
        asm volatile(LOCK_PREFIX "xorb %1,%0"
            : CONST_MASK_ADDR(nr, addr)
            : "iq" ((u8)CONST_MASK(nr)));
    } else {
        asm volatile(LOCK_PREFIX "btc %1,%0"
            : BITOP_ADDR(addr)
            : "Ir" (nr));
    }
}
```

它和 `set_bit` 函数很相似，但也存在两点不同。第一处差异为 `xor` 操作而不是 `or` 。第二处差异为 `btc` （LCTT 译注：原文为 `bts` ，为作者笔误） 而不是 `bts` 。

目前，我们了解了最重要的体系特定的位数组操作，是时候看看一般的位图 API 了。

通用位操作

除了 arch/x86/include/asm/bitops.h 中体系特定的 API 外，Linux 内核提供了操作位数组的通用 API。正如我们本部分开头所了解的一样，我们可以在 include/linux/bitmap.h 头文件和 lib/bitmap.c 源文件中找到它。但在查看这些源文件之前，我们先看看 include/linux/bitops.h 头文件，其提供了一系列有用的宏，让我们看看它们当中一部分。

首先我们看看以下 4 个 宏：

- `for_each_set_bit`
- `for_each_set_bit_from`
- `for_each_clear_bit`
- `for_each_clear_bit_from`

所有这些宏都提供了遍历位数组中某些位集合的迭代器。第一个宏迭代那些被置位的位。第二个宏也是一样，但它是从某一个确定的位开始。最后两个宏做的一样，但是迭代那些被清位的位。让我们看看 `for_each_set_bit` 宏：

```
#define for_each_set_bit(bit, addr, size) \
    for ((bit) = find_first_bit((addr), (size));        \
         (bit) < (size);                         \
         (bit) = find_next_bit((addr), (size), (bit) + 1))
```

正如我们所看到的，它使用了三个参数，并展开为一个循环，该循环从作为 `find_first_bit` 函数返回结果的第一个置位开始，到小于给定大小的最后一个置位为止。

除了这四个宏，arch/x86/include/asm/bitops.h 也提供了 `64-bit` 或 `32-bit` 变量循环的 API 等等。

下一个 头文件 提供了操作位数组的 API。例如，它提供了以下两个函数：

- `bitmap_zero`；
- `bitmap_fill`.

它们分别可以清除一个位数组和用 `1` 填充位数组。让我们看看 `bitmap_zero` 函数的实现：

```
static inline void bitmap_zero(unsigned long *dst, unsigned int nbits)
{
    if (small_const_nbits(nbits))
        *dst = 0UL;
    else {
        unsigned int len = BITS_TO_LONGS(nbits) * sizeof(unsigned long);
        memset(dst, 0, len);
    }
}
```

首先我们可以看到对 `nbits` 的检查。 `small_const_nbits` 是一个定义在同一个头文件 的宏：

```
#define small_const_nbits(nbits) \
    (__builtin_constant_p(nbits) && (nbits) <= BITS_PER_LONG)
```

正如我们可以看到的，它检查 `nbits` 是否为编译期已知常量，并且其值不超过 `BITS_PER_LONG` 或 `64` 。如果位数目没有超过一个 `long` 变量的位数，我们可以仅仅设置为 0。在其他情况，我们需要计算有多少个需要填充位数组的 `long` 变量并且使用 memset 进行填充。

`bitmap_fill` 函数的实现和 `biramp_zero` 函数很相似，除了我们需要在给定的位数组中填写 `0xff` 或 `0b11111111` ：

```
static inline void bitmap_fill(unsigned long *dst, unsigned int nbits)
{
    unsigned int nlongs = BITS_TO_LONGS(nbits);
    if (!small_const_nbits(nbits)) {
        unsigned int len = (nlongs - 1) * sizeof(unsigned long);
        memset(dst, 0xff,  len);
    }
    dst[nlongs - 1] = BITMAP_LAST_WORD_MASK(nbits);
}
```

除了 `bitmap_fill` 和 `bitmap_zero`，include/linux/bitmap.h 头文件也提供了和 `bitmap_zero` 很相似的 `bitmap_copy`，只是仅仅使用 memcpy 而不是 memset 这点差异而已。它也提供了位数组的按位操作，像 `bitmap_and`, `bitmap_or`, `bitamp_xor` 等等。我们不会探讨这些函数的实现了，因为如果你理解了本部分的所有内容，这些函数的实现是很容易理解的。无论如何，如果你对这些函数是如何实现的感兴趣，你可以打开并研究 include/linux/bitmap.h 头文件。

本部分到此为止。

注： 本文由 LCTT 原创翻译，Linux中国 荣誉推出

链接

- bitmap
- linked data structures
- tree data structures
- hot-plug
- cpumasks
- IRQs
- API
- atomic operations
- xchg instruction
- cmpxchg instruction
- lock instruction
- bts instruction
- btr instruction
- bt instruction
- sbb instruction
- btc instruction
- man memcpy
- man memset
- CF
- inline assembler
- gcc

理论

这一章描述各种理论性概念和那些不直接涉及实践，但是知道了会很有用的概念。

分页

简介

在 Linux 内核启动过程中的第五部分，我们学到了内核在启动的最早阶段都做了哪些工作。接下来，在我们明白内核如何运行第一个 init 进程之前，内核初始化其他部分，比如加载 `initrd` ，初始化 lockdep ，以及许多许多其他的工作。

是的，那将有很多不同的事，但是还有更多更多更多关于内存的工作。

在我看来，一般而言，内存管理是 Linux 内核和系统编程最复杂的部分之一。这就是为什么在我们学习内核初始化过程之前，需要了解分页。

分页是将线性地址转换为物理地址的机制。如果我们已经读过了这本书之前的部分，你可能记得我们在实模式下有分段机制，当时物理地址是由左移四位段寄存器加上偏移算出来的。我们也看了保护模式下的分段机制，其中我们使用描述符表得到描述符，进而得到基地址，然后加上偏移地址就获得了实际物理地址。由于我们在 64 位模式，我们将看分页机制。

正如 Intel 手册中说的：

> 分页机制提供一种机制，为了实现常见的按需分页，比如虚拟内存系统就是将一个程序执行环境中的段按照需求被映射到物理地址。

所以... 在这个帖子中我将尝试解释分页背后的理论。当然它将与64位版本的 Linux 内核关系密切，但是我们将不会深入太多细节（至少在这个帖子里面）。

开启分页

有三种分页模式：

- 32 位分页模式；
- PAE 分页；
- IA-32e 分页。

我们这里将只解释最后一种模式。为了开启 `IA-32e 分页模式` ，我们需要做如下事情：

- 设置 `CR0.PG` 位；
- 设置 `CR4.PAE` 位；
- 设置 `IA32_EFER.LME` 位。

我们已经在 arch/x86/boot/compressed/head_64.S 中看见了这些位被设置了：

```
movl    $(X86_CR0_PG | X86_CR0_PE), %eax
movl    %eax, %cr0
```

and

```
movl    $MSR_EFER, %ecx
rdmsr
btsl    $_EFER_LME, %eax
wrmsr
```

分页数据结构

分页将线性地址分为固定尺寸的页。页会被映射进入物理地址空间或外部存储设备。这个固定尺寸在 `x86_64` 内核中是 `4096` 字节。为了将线性地址转换位物理地址，需要使用到一些特殊的数据结构。每个结构都是 `4096` 字节并包含 `512` 项（这只为 `PAE` 和 `IA32_EFER.LME` 模式）。分页结构是层次级的，Linux 内核在 `x86_64` 框架中使用4层的分层机制。CPU使用一部分线性地址去确定另一个分页结构中的项，这个分页结构可能在最低层，物理内存区域（页框），在这个区域的物理地址（页偏移）。最高层的分页结构的地址存储在 `cr3` 寄存器中。我们已经从 arch/x86/boot/compressed/head_64.S 这个文件中已经看到了。

```
leal    pgtable(%ebx), %eax
movl    %eax, %cr3
```

我们构建页表结构并且将这个最高层结构的地址存放在 `cr3` 寄存器中。这里 `cr3` 用于存储最高层结构的地址，在 Linux 内核中被称为 `PML4` 或 `Page Global Directory` 。 `cr3` 是一个64位的寄存器，并且有着如下的结构：

```
63                  52 51                                                    32
 --------------------------------------------------------------------------------
|                     |                                                          |
|    Reserved MBZ     |            Address of the top level structure            |
|                     |                                                          |
 --------------------------------------------------------------------------------
31                                  12 11              5     4     3 2             0
 --------------------------------------------------------------------------------
|                                     |               |  P  |  P  |              |
|  Address of the top level structure |   Reserved    |  C  |  W  |   Reserved   |
|                                     |               |  D  |  T  |              |
 --------------------------------------------------------------------------------
```

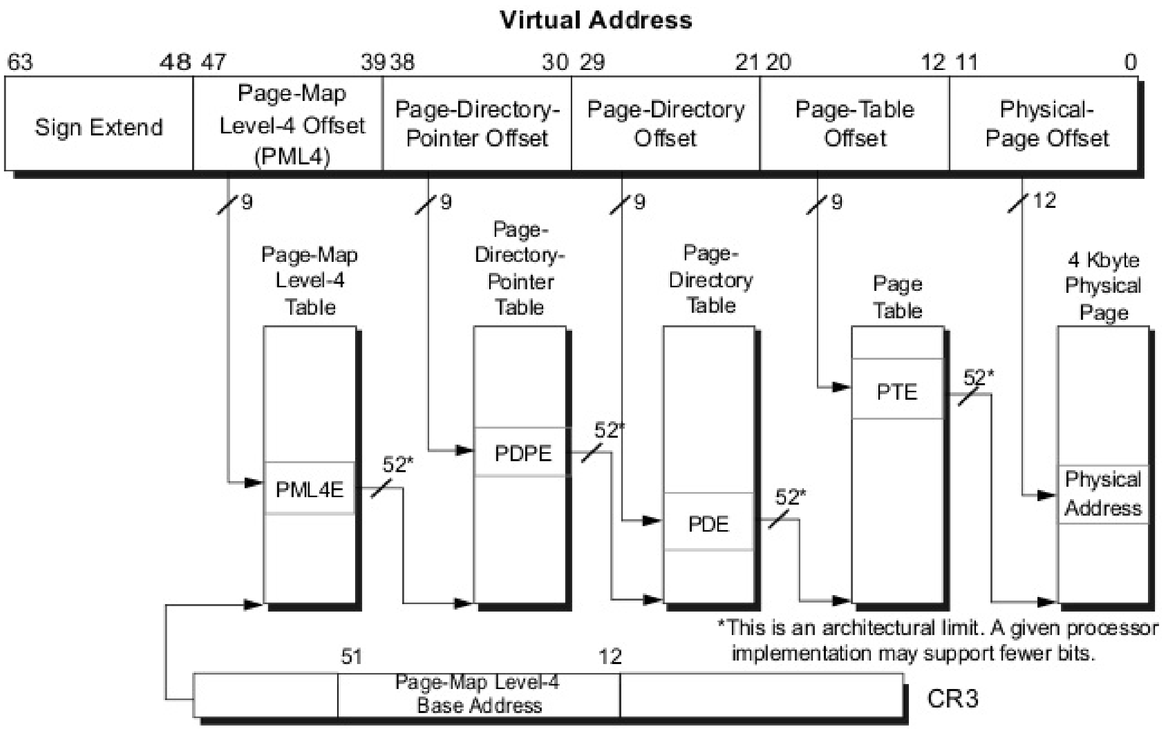
这些字段有着如下的意义：

- 第 0 到第 2 位 - 忽略；
- 第 12 位到第 51 位 - 存储最高层分页结构的地址；
- 第 3 位 到第 4 位 - PWT 或 Page-Level Writethrough 和 PCD 或 Page-level Cache Disable 显示。这些位控制页或者页表被硬件缓存处理的方式；
- 保留位 - 保留，但必须为 0 ；
- 第 52 到第 63 位 - 保留，但必须为 0 ；

线性地址转换过程如下所示：

- 一个给定的线性地址传递给 MMU 而不是存储器总线；
- 64位线性地址分为很多部分。只有低 48 位是有意义的，它意味着 `2^48` 或 256TB 的线性地址空间在任意给定时间内都可以被访问；
- `cr3` 寄存器存储这个最高层分页数据结构的地址；
- 给定的线性地址中的第 39 位到第 47 位存储一个第 4 级分页结构的索引，第 30 位到第 38 位存储一个第3级分页结构的索引，第 29 位到第 21 位存储一个第 2 级分页结构的索引，第 12 位到第 20 位存储一个第 1 级分页结构的索引，第 0 位到第 11 位提供物理页的字节偏移；

按照图示，我们可以这样想象它：



每一个对线性地址的访问不是一个管态访问就是用户态访问。这个访问是被 `CPL (Current Privilege Level)` 所决定。如果 `CPL < 3` ，那么它是管态访问级，否则，它就是用户态访问级。比如，最高级页表项包含访问位和如下的结构：

```
63  62                  52 51                                                    32
 --------------------------------------------------------------------------------
| N |                   |                                                        |
|   |     Available     |     Address of the paging structure on lower level   |
| X |                   |                                                        |
 --------------------------------------------------------------------------------
31                                              12 11  9 8 7 6 5   4   3 2 1     0
 --------------------------------------------------------------------------------
|                                                |     | M |I| | P | P |U|W|     |
| Address of the paging structure on lower level | AVL | B |G|A| C | W | | |  P |
|                                                |     | Z |N| | D | T |S|R|     |
 --------------------------------------------------------------------------------
```

其中：

- 第 63 位 - N/X 位（不可执行位）显示被这个页表项映射的所有物理页执行代码的能力；
- 第 52 位到第 62 位 - 被CPU忽略，被系统软件使用；
- 第 12 位到第 51 位 - 存储低级分页结构的物理地址；
- 第 9 位到第 11 位 - 被 CPU 忽略；
- MBZ - 必须为 0 ；
- 忽略位；
- A - 访问位暗示物理页或者页结构被访问；
- PWT 和 PCD 用于缓存；
- U/S - 用户/管理位控制对被这个页表项映射的所有物理页用户访问；
- R/W - 读写位控制着被这个页表项映射的所有物理页的读写权限
- P - 存在位。当前位表示页表或物理页是否被加载进内存；

好的，我们知道了分页结构和它们的表项。现在我们来看一下 Linux 内核中的 4 级分页机制的一些细节。

Linux 内核中的分页结构

就如我们已经看到的那样， `x86_64` Linux 内核使用4级页表。它们的名字是：

- 全局页目录
- 上层页目录
- 中间页目录
- 页表项

在你已经编译和安装 Linux 内核之后，你可以看到保存了内核函数的虚拟地址的文件 `System.map` 。例如：

```
$ grep "start_kernel" System.map
ffffffff81efe497 T x86_64_start_kernel
ffffffff81efeaa2 T start_kernel
```

这里我们可以看见 `0xffffffff81efe497` 。我怀疑你是否真的有安装这么多内存。但是无论如何， `start_kernel` 和 `x86_64_start_kernel` 将会被执行。在 `x86_64` 中，地址空间的大小是 `2^64` ，但是它太大了，这就是为什么我们使用一个较小的地址空间，只是 48 位的宽度。所以一个情况出现，虽然物理地址空间限制到 48 位，但是寻址仍然使用 64 位指针。 这个问题是如何解决的？看下面的这个表。

```
0xffffffffffffffff  +-----------+
                    |           |
                    |           | Kernelspace
                    |           |
 0xffff800000000000 +-----------+
                    |           |
                    |           |
                    |   hole    |
                    |           |
                    |           |
0x00007fffffffffff  +-----------+
                    |           |
                    |           |  Userspace
                    |           |
0x0000000000000000  +-----------+
```

这个解决方案是 `sign extension` 。这里我们可以看到一个虚拟地址的低 48 位可以被用于寻址。第 48 位到第 63 位全是 0 或 1 。注意这个虚拟地址空间被分为两部分：

- 内核空间
- 用户空间

用户空间占用虚拟地址空间的低部分，从 `0x000000000000000` 到 `0x00007fffffffffff` ，而内核空间占据从 `0xffff8000000000` 到 `0xffffffffffffffff` 的高部分。注意，第 48 位到第 63 位是对于用户空间是 0 ，对于内核空间是 1 。内核空间和用户空间中的所有地址是标准地址，而在这些内存区域中间有非标准区域。这两块内存区域（内核空间和用户空间）合起来是 48 位宽度。我们可以在 Documentation/x86/x86_64/mm.txt 找到 4 级页表下的虚拟内存映射:

```
0000000000000000 - 00007fffffffffff (=47 bits) user space, different per mm
hole caused by [48:63] sign extension
ffff800000000000 - ffff87ffffffffff (=43 bits) guard hole, reserved for hypervisor
ffff880000000000 - ffffc7ffffffffff (=64 TB) direct mapping of all phys. memory
ffffc80000000000 - ffffc8ffffffffff (=40 bits) hole
ffffc90000000000 - ffffe8ffffffffff (=45 bits) vmalloc/ioremap space
ffffe90000000000 - ffffe9ffffffffff (=40 bits) hole
ffffea0000000000 - ffffeaffffffffff (=40 bits) virtual memory map (1TB)
... unused hole ...
ffffec0000000000 - fffffc0000000000 (=44 bits) kasan shadow memory (16TB)
... unused hole ...
ffffff0000000000 - ffffff7fffffffff (=39 bits) %esp fixup stacks
... unused hole ...
ffffffff80000000 - ffffffffa0000000 (=512 MB)  kernel text mapping, from phys 0
ffffffffa0000000 - ffffffffff5fffff (=1525 MB) module mapping space
ffffffffff600000 - ffffffffffdfffff (=8 MB) vsyscalls
ffffffffffe00000 - ffffffffffffffff (=2 MB) unused hole
```

这里我们可以看到用户空间，内核空间和非标准空间的内存映射。用户空间的内存映射很简单。让我们来更近地查看内核空间。我们可以看到它始于为管理程序 (hypervisor) 保留的防御空洞 (guard hole) 。我们可以在 arch/x86/include/asm/page_64_types.h 这个文件中看到防御空洞的概念！

```
#define __PAGE_OFFSET _AC(0xffff880000000000, UL)
```

以前防御空洞和 `__PAGE_OFFSET` 是从 `0xffff800000000000` 到 `0xffff80ffffffffff` ，用来防止对非标准区域的访问，但是后来为了管理程序扩展了 3 位。

紧接着是内核空间中最低的可用空间 - `ffff880000000000` 。这个虚拟地址空间是为了所有的物理内存的直接映射。在这块空间之后，还是防御空洞。它位于所有物理内存的直接映射地址和被 vmalloc 分配的地址之间。在第一个 1TB 的虚拟内存映射和无用的空洞之后，我们可以看到 `ksan` 影子内存 (shadow memory) 。它是通过 commit 提交到内核中，并且保持内核空间无害。在紧接着的无用空洞之后，我们可以看到 `esp` 固定栈（我们会在本书其他部分讨论它）。内核代码段的开始从物理地址 - `0` 映射。我们可以在相同的文件中找到将这个地址定义为 `__PAGE_OFFSET` 。

```
#define __START_KERNEL_map      _AC(0xffffffff80000000, UL)
```

通常内核的 `.text` 段开始于 `CONFIG_PHYSICAL_START` 偏移。我们已经在 ELF64 相关帖子中看见。

```
readelf -s vmlinux | grep ffffffff81000000
     1: ffffffff81000000     0 SECTION LOCAL  DEFAULT    1
 65099: ffffffff81000000     0 NOTYPE  GLOBAL DEFAULT    1 _text
 90766: ffffffff81000000     0 NOTYPE  GLOBAL DEFAULT    1 startup_64
```

这里我将 `CONFIG_PHYSICAL_START` 设置为 `0x1000000` 来检查 `vmlinux` 。所以我们有内核代码段的起始点 - `0xffffffff80000000` 和 偏移 - `0x1000000` ，计算出来的虚拟地址将会是 `0xffffffff80000000 + 1000000 = 0xffffffff81000000` 。

在内核代码段之后有一个为内核模块 `vsyscalls` 准备的虚拟内存区域和 2M 无用的空洞。

我们已经看见内核虚拟内存映射是如何布局的以及虚拟地址是如何转换位物理地址。让我们以下面的地址为例：

```
0xffffffff81000000
```

在二进制内它将是：

```
1111111111111111 111111111 111111110 000001000 000000000 000000000000
      63:48          47:39     38:30     29:21     20:12      11:0
```

这个虚拟地址将被分为如下描述的几部分：

- `48-63` - 不使用的位；
- `37-49` - 给定线性地址的这些位描述一个 4 级分页结构的索引；
- `30-38` - 这些位存储一个 3 级分页结构的索引；
- `21-29` - 这些位存储一个 2 级分页结构的索引；
- `12-20` - 这些位存储一个 1 级分页结构的索引；
- `0-11` - 这些位提供物理页的偏移；

就这样了。现在你知道了一些关于分页理论，而且我们可以在内核源码上更近一步，查看那些最先的初始化步骤。

总结

这简短的关于分页理论的部分至此已经结束了。当然，这个帖子不可能包含分页的所有细节，但是我们很快会看到在实践中 Linux 内核如何构建分页结构以及使用它们工作。

链接

- [Paging on Wikipedia]

- Intel 64 and IA-32 architectures software developer's manual volume 3A
- MMU
- ELF64
- Documentation/x86/x86_64/mm.txt
- Last part - Kernel booting process

ELF文件格式

ELF (Executable and Linkable Format)是一种为可执行文件，目标文件，共享链接库和内核转储(core dumps)准备的标准文件格式。 Linux和很多类Unix操作系统都使用这个格式。 让我们来看一下64位ELF文件格式的结构以及内核源码中有关于它的一些定义。

一个ELF文件由以下三部分组成：

- ELF头(ELF header) - 描述文件的主要特性：类型，CPU架构，入口地址，现有部分的大小和偏移等等；
- 程序头表(Program header table) - 列举了所有有效的段(segments)和他们的属性。 程序头表需要加载器将文件中的节加载到虚拟内存段中；
- 节头表(Section header table) - 包含对节(sections)的描述。

现在让我们对这些部分有一些更深的了解。

ELF头(ELF header)

ELF头(ELF header)位于文件的开始位置。 它的主要目的是定位文件的其他部分。 文件头主要包含以下字段：

- ELF文件鉴定 - 一个字节数组用来确认文件是否是一个ELF文件，并且提供普通文件特征的信息；
- 文件类型 - 确定文件类型。 这个字段描述文件是一个重定位文件，或可执行文件,或...；
- 目标结构；
- ELF文件格式的版本；
- 程序入口地址；
- 程序头表的文件偏移；
- 节头表的文件偏移；
- ELF头(ELF header)的大小；
- 程序头表的表项大小；
- 其他字段...

你可以在内核源码种找到表示ELF64 header的结构体 `elf64_hdr` ：

```
typedef struct elf64_hdr {
    unsigned char    e_ident[EI_NIDENT];
    Elf64_Half e_type;
    Elf64_Half e_machine;
    Elf64_Word e_version;
    Elf64_Addr e_entry;
    Elf64_Off e_phoff;
    Elf64_Off e_shoff;
    Elf64_Word e_flags;
    Elf64_Half e_ehsize;
    Elf64_Half e_phentsize;
    Elf64_Half e_phnum;
    Elf64_Half e_shentsize;
    Elf64_Half e_shnum;
    Elf64_Half e_shstrndx;
} Elf64_Ehdr;
```

这个结构体定义在 elf.h

节(sections)

所有的数据都存储在ELF文件的节(sections)中。 我们通过节头表中的索引(index)来确认节(sections)。 节头表表项包含以下字段：

- 节的名字；
- 节的类型；
- 节的属性；
- 内存地址；
- 文件中的偏移；
- 节的大小；
- 到其他节的链接；
- 各种各样的信息；
- 地址对齐；
- 这个表项的大小，如果有的话；

而且，在linux内核中结构体 `elf64_shdr` 如下所示:

```
typedef struct elf64_shdr {
    Elf64_Word sh_name;
    Elf64_Word sh_type;
    Elf64_Xword sh_flags;
    Elf64_Addr sh_addr;
    Elf64_Off sh_offset;
    Elf64_Xword sh_size;
    Elf64_Word sh_link;
    Elf64_Word sh_info;
    Elf64_Xword sh_addralign;
    Elf64_Xword sh_entsize;
} Elf64_Shdr;
```

elf.h

程序头表**(Program header table)**

在可执行文件或者共享链接库中所有的节(sections)都被分为多个段(segments)。 程序头是一个结构的数组，每一个结构都表示一个段(segments)。 它的结构就像这样：

```
typedef struct elf64_phdr {
    Elf64_Word p_type;
    Elf64_Word p_flags;
    Elf64_Off p_offset;
    Elf64_Addr p_vaddr;
    Elf64_Addr p_paddr;
    Elf64_Xword p_filesz;
    Elf64_Xword p_memsz;
    Elf64_Xword p_align;
} Elf64_Phdr;
```

在内核源码中。

`elf64_phdr` 定义在相同的 elf.h 文件中.

EFL文件也包含其他的字段或结构。 你可以在 Documentation 中查看。 现在我们来查看一下 `vmlinux` 这个ELF文件。

vmlinux

`vmlinux` 也是一个可重定位的ELF文件。 我们可以使用 `readelf` 工具来查看它。 首先，让我们看一下它的头部：

```
$ readelf -h  vmlinux
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1000000
  Start of program headers:          64 (bytes into file)
  Start of section headers:          381608416 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         5
  Size of section headers:           64 (bytes)
  Number of section headers:         73
  Section header string table index: 70
```

我们可以看出 `vmlinux` 是一个64位可执行文件。 我们可以从 Documentation/x86/x86_64/mm.txt 读到相关信息:

```
ffffffff80000000 - ffffffffa0000000 (=512 MB)  kernel text mapping, from phys 0
```

之后我们可以在 `vmlinux` ELF文件中查看这个地址：

```
$ readelf -s vmlinux | grep ffffffff81000000
     1: ffffffff81000000     0 SECTION LOCAL  DEFAULT    1
 65099: ffffffff81000000     0 NOTYPE  GLOBAL DEFAULT    1 _text
 90766: ffffffff81000000     0 NOTYPE  GLOBAL DEFAULT    1 startup_64
```

值得注意的是，`startup_64` 例程的地址不是 `ffffffff80000000`，而是 `ffffffff81000000`。现在我们来解释一下。

我们可以在 arch/x86/kernel/vmlinux.lds.S 看见如下的定义：

```
    . = __START_KERNEL;
    ...
    ...
    ..
    /* Text and read-only data */
    .text :  AT(ADDR(.text) - LOAD_OFFSET) {
        _text = .;
        ...
        ...
        ...
    }
```

其中，`__START_KERNEL` 定义如下:

```
#define __START_KERNEL        (__START_KERNEL_map + __PHYSICAL_START)
```

从这个文档中看出，`__START_KERNEL_map` 的值是 `ffffffff80000000` 以及 `__PHYSICAL_START` 的值是 `0x1000000` 。这就是 `startup_64` 的地址是 `ffffffff81000000` 的原因了。

最后我们通过以下命令来得到程序头表的内容：

```
readelf -l vmlinux

Elf file type is EXEC (Executable file)
Entry point 0x1000000
There are 5 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000200000 0xffffffff81000000 0x0000000001000000
                 0x0000000000cfd000 0x0000000000cfd000  R E    200000
  LOAD           0x0000000001000000 0xffffffff81e00000 0x0000000001e00000
                 0x0000000000100000 0x0000000000100000  RW     200000
  LOAD           0x0000000001200000 0x0000000000000000 0x0000000001f00000
                 0x0000000000014d98 0x0000000000014d98  RW     200000
  LOAD           0x0000000001315000 0xffffffff81f15000 0x0000000001f15000
                 0x000000000011d000 0x0000000000279000  RWE    200000
  NOTE           0x0000000000b17284 0xffffffff81917284 0x0000000001917284
                 0x0000000000000024 0x0000000000000024         4

 Section to Segment mapping:
  Segment Sections...
   00     .text .notes __ex_table .rodata __bug_table .pci_fixup .builtin_fw
          .tracedata __ksymtab __ksymtab_gpl __kcrctab __kcrctab_gpl
          __ksymtab_strings __param __modver
   01     .data .vvar
   02     .data..percpu
   03     .init.text .init.data .x86_cpu_dev.init .altinstructions
          .altinstr_replacement .iommu_table .apicdrivers .exit.text
          .smp_locks .data_nosave .bss .brk
```

这里我们可以看出五个包含节(sections)列表的段(segments)。 你可以在生成的链接器脚本 - `arch/x86/kernel/vmlinux.lds` 中找到所有的节(sections)。

就这样吧。 当然，它不是ELF(Executable and Linkable Format)的完整描述，但是如果你想要知道更多，可以参考这个文档 - 这里

Inline assembly

Introduction

While reading source code in the Linux kernel, I often see statements like this:

```
__asm__("andq %%rsp,%0; ":"=r" (ti) : "0" (CURRENT_MASK));
```

Yes, this is inline assembly or in other words assembler code which is integrated in a high level programming language. In this case the high level programming language is C. Yes, the `C` programming language is not very high-level, but still.

If you are familiar with the assembly programming language, you may notice that `inline assembly` is not very different from normal assembler. Moreover, the special form of inline assembly which is called `basic form` is exactly the same. For example:

```
__asm__("movq %rax, %rsp");
```

or:

```
__asm__("hlt");
```

The same code (of course without `__asm__` prefix) you might see in plain assembly code. Yes, this is very similar, but not so simple as it might seem at first glance. Actually, the GCC supports two forms of inline assembly statements:

- `basic` ;
- `extended` .

The basic form consists of only two things: the `__asm__` keyword and the string with valid assembler instructions. For example it may look something like this:

```
__asm__("movq    $3, %rax\t\n"
        "movq    %rsi, %rdi");
```

The `asm` keyword may be used in place of `__asm__` , however `__asm__` is portable whereas the `asm` keyword is a `GNU` extension. In further examples I will only use the `__asm__` variant.

If you know assembly programming language this looks pretty familiar. The main problem is in the second form of inline assembly statements - `extended`. This form allows us to pass parameters to an assembly statement, perform jumps etc. Does not sound difficult, but requires knowledge of special rules in addition to knowledge of the assembly language. Every time I see yet another piece of inline assembly code in the Linux kernel, I need to refer to the official documentation of `GCC` to remember how a particular `qualifier` behaves or what the meaning of `=&r` is for example.

I've decided to write this part to consolidate my knowledge related to the inline assembly, as inline assembly statements are quite common in the Linux kernel and we may see them in linux-insides parts sometimes. I thought that it would be useful if we have a special part which contains information on more important aspects of the inline assembly. Of course you may find comprehensive information about inline assembly in the official documentation, but I like to put everything in one place.

Note: This part will not provide guide for assembly programming. It is not intended to teach you to write programs with assembler or to know what one or another assembler instruction means. Just a little memo for extended asm.

Introduction to extended inline assembly

So, let's start. As I already mentioned above, the `basic` assembly statement consists of the `asm` or `__asm__` keyword and set of assembly instructions. This form is in no way different from "normal" assembly. The most interesting part is inline assembler with operands, or `extended` assembler. An extended assembly statement looks more complicated and consists of more than two parts:

```
__asm__ [volatile] [goto] (AssemblerTemplate
                           [ : OutputOperands ]
                           [ : InputOperands  ]
                           [ : Clobbers       ]
                           [ : GotoLabels     ]);
```

All parameters which are marked with squared brackets are optional. You may notice that if we skip the optional parameters and the modifiers `volatile` and `goto` we obtain the `basic` form.

Let's start to consider this in order. The first optional `qualifier` is `volatile`. This specifier tells the compiler that an assembly statement may produce `side effects`. In this case we need to prevent compiler optimizations related to the given assembly statement. In simple

terms the `volatile` specifier instructs the compiler not to modify the statement and place it exactly where it was in the original code. As an example let's look at the following function from the Linux kernel:

```
static inline void native_load_gdt(const struct desc_ptr *dtr)
{
    asm volatile("lgdt %0"::"m" (*dtr));
}
```

Here we see the `native_load_gdt` function which loads a base address from the Global Descriptor Table to the `GDTR` register with the `lgdt` instruction. This assembly statement is marked with `volatile` qualifier. It is very important that the compiler does not change the original place of this assembly statement in the resulting code. Otherwise the `GDTR` register may contain wrong address for the `Global Descriptor Table` or the address may be correct, but the structure has not been filled yet. This can lead to an exception being generated, preventing the kernel from booting correctly.

The second optional `qualifier` is the `goto`. This qualifier tells the compiler that the given assembly statement may perform a jump to one of the labels which are listed in the `GotoLabels`. For example:

```
__asm__ goto("jmp %l[label]" : : : : label);
```

Since we finished with these two qualifiers, let's look at the main part of an assembly statement body. As we have seen above, the main part of an assembly statement consists of the following four parts:

- set of assembly instructions;
- output parameters;
- input parameters;
- clobbers.

The first represents a string which contains a set of valid assembly instructions which may be separated by the `\t\n` sequence. Names of processor registers must be prefixed with the `%%` sequence in `extended` form and other symbols like immediates must start with the `$` symbol. The `OutputOperands` and `InputOperands` are comma-separated lists of C variables which may be provided with "constraints" and the `Clobbers` is a list of registers or other values which are modified by the assembler instructions from the `AssemblerTemplate` beyond those listed in the `OutputOperands`. Before we dive into the examples we have to know a little bit about `constraints`. A constraint is a string which specifies placement of an operand. For example the value of an operand may be written to a processor register or read from memory etc.

Consider the following simple example:

```
#include <stdio.h>

int main(void)
{
        unsigned long a = 5;
        unsigned long b = 10;
        unsigned long sum = 0;

        __asm__("addq %1,%2" : "=r" (sum) : "r" (a), "0" (b));
        printf("a + b = %lu\n", sum);
        return 0;
}
```

Let's compile and run it to be sure that it works as expected:

```
$ gcc test.c -o test
./test
a + b = 15
```

Ok, great. It works. Now let's look at this example in detail. Here we see a simple `C` program which calculates the sum of two variables placing the result into the `sum` variable and in the end we print the result. This example consists of three parts. The first is the assembly statement with the add instruction. It adds the value of the source operand together with the value of the destination operand and stores the result in the destination operand. In our case:

```
addq %1, %2
```

will be expanded to the:

```
addq a, b
```

Variables and expressions which are listed in the `OutputOperands` and `InputOperands` may be matched in the `AssemblerTemplate`. An input/output operand is designated as `%N` where the `N` is the number of operand from left to right beginning from `zero`. The second part of the our assembly statement is located after the first `:` symbol and contains the definition of the output value:

```
"=r" (sum)
```

Notice that the `sum` is marked with two special symbols: `=r`. This is the first constraint that we have encountered. The actual constraint here is only `r` itself. The `=` symbol is `modifier` which denotes output value. This tells to compiler that the previous value will be discarded and replaced by the new data. Besides the `=` modifier, `GCC` provides support for following three modifiers:

- `+` - an operand is read and written by an instruction;
- `&` - output register shouldn't overlap an input register and should be used only for output;
- `%` - tells the compiler that operands may be commutative.

Now let's go back to the `r` qualifier. As I mentioned above, a qualifier denotes the placement of an operand. The `r` symbol means a value will be stored in one of the general purpose register. The last part of our assembly statement:

```
"r" (a), "0" (b)
```

These are input operands - variables `a` and `b`. We already know what the `r` qualifier does. Now we can have a look at the constraint for the variable `b`. The `0` or any other digit from `1` to `9` is called "matching constraint". With this a single operand can be used for multiple roles. The value of the constraint is the source operand index. In our case `0` will match `sum`. If we look at assembly output of our program:

```
0000000000400400 <main>:
  ...
  ...
  ...
  4004fe:       48 c7 45 f8 05 00 00    movq   $0x5,-0x8(%rbp)
  400506:       48 c7 45 f0 0a 00 00    movq   $0xa,-0x10(%rbp)

  400516:       48 8b 55 f8             mov    -0x8(%rbp),%rdx
  40051a:       48 8b 45 f0             mov    -0x10(%rbp),%rax
  40051e:       48 01 d0                add    %rdx,%rax
```

First of all our values `5` and `10` will be put at the stack and then these values will be moved to the two general purpose registers: `%rdx` and `%rax`.

This way the `%rax` register is used for storing the value of the `b` as well as storing the result of the calculation. **NOTE** that I've used `gcc 6.3.1` version, so the resulted code of your compiler may differ.

We have looked at input and output parameters of an inline assembly statement. Before we move on to other constraints supported by `gcc`, there is one remaining part of the inline assembly statement we have not discussed yet - `clobbers`.

Clobbers

As mentioned above, the "clobbered" part should contain a comma-separated list of registers whose content will be modified by the assembler code. This is useful if our assembly expression needs additional registers for calculation. If we add clobbered registers to the inline assembly statement, the compiler take this into account and the register in question will not simultaneously be used by the compiler.

Consider the example from before, but we will add an additional, simple assembler instruction:

```
__asm__("movq $100, %%rdx\t\n"
        "addq %1,%2" : "=r" (sum) : "r" (a), "0" (b));
```

If we look at the assembly output:

```
0000000000400400 <main>:
  ...
  ...
  ...
  4004fe:       48 c7 45 f8 05 00 00    movq   $0x5,-0x8(%rbp)
  400506:       48 c7 45 f0 0a 00 00    movq   $0xa,-0x10(%rbp)

  400516:       48 8b 55 f8             mov    -0x8(%rbp),%rdx
  40051a:       48 8b 45 f0             mov    -0x10(%rbp),%rax

  40051e:       48 c7 c2 64 00 00 00    mov    $0x64,%rdx
  400525:       48 01 d0                add    %rdx,%rax
```

we will see that the `%rdx` register is overwritten with `0x64` or `100` and the result will be `110` instead of `10` . Now if we add the `%rdx` register to the list of `clobbered` registers:

```
__asm__("movq $100, %%rdx\t\n"
        "addq %1,%2" : "=r" (sum) : "r" (a), "0" (b) : "%rdx");
```

and look at the assembler output again:

```
0000000000400400 <main>:
  4004fe:       48 c7 45 f8 05 00 00    movq   $0x5,-0x8(%rbp)
  400506:       48 c7 45 f0 0a 00 00    movq   $0xa,-0x10(%rbp)

  400516:       48 8b 4d f8             mov    -0x8(%rbp),%rcx
  40051a:       48 8b 45 f0             mov    -0x10(%rbp),%rax

  40051e:       48 c7 c2 64 00 00 00    mov    $0x64,%rdx
  400525:       48 01 c8                add    %rcx,%rax
```

the `%rcx` register will be used for `sum` calculation, preserving the intended semantics of the program. Besides general purpose registers, we may pass two special specifiers. They are:

- `cc` ;
- `memory` .

The first - `cc` indicates that an assembler code modifies flags register. This is typically used if the assembly within contains arithmetic or logic instructions:

```
__asm__("incq %0" ::""(variable): "cc");
```

The second `memory` specifier tells the compiler that the given inline assembly statement executes read/write operations on memory not specified by operands in the output list. This prevents the compiler from keeping memory values loaded and cached in registers. Let's take a look at the following example:

```
#include <stdio.h>

int main(void)
{
        unsigned long a[3] = {10000000000, 0, 1};
        unsigned long b = 5;

        __asm__ volatile("incq %0" :: "m" (a[0]));

        printf("a[0] - b = %lu\n", a[0] - b);
        return 0;
}
```

This example may be artificial, but it illustrates the main idea. Here we have an array of integers and one integer variable. The example is pretty simple, we take the first element of `a` and increment its value. After this we subtract the value of `b` from the first element of `a` . In the end we print the result. If we compile and run this simple example the result may surprise you:

```
~$ gcc -O3  test.c -o test
~$ ./test
a[0] - b = 9999999995
```

The result is `a[0] - b = 9999999995` here, but why? We incremented `a[0]` and subtracted `b`, so the result should be `a[0] - b = 9999999996` here.

If we have a look at the assembler output for this example:

```
00000000004004f6 <main>:
  4004b4:       48 b8 00 e4 0b 54 02    movabs $0x2540be400,%rax
  4004be:       48 89 04 24             mov    %rax,(%rsp)
  ...
  ...
  ...
  40050e:       ff 44 24 f0             incq   (%rsp)

  4004d8:       48 be fb e3 0b 54 02    movabs $0x2540be3fb,%rsi
```

we will see that the first element of the `a` contains the value `0x2540be400` (`10000000000`). The last two lines of code are the actual calculations.

We see our increment instruction with `incq` but then just a move of `0x2540be3fb` (`9999999995`) to the `%rsi` register. This looks strange.

The problem is we have passed the `-O3` flag to `gcc`, so the compiler did some constant folding and propagation to determine the result of `a[0] - 5` at compile time and reduced it to a `movabs` with a constant `0x2540be3fb` or `9999999995` in runtime.

Let's now add `memory` to the clobbers list:

```
__asm__ volatile("incq %0" :: "m" (a[0]) : "memory");
```

and the new result of running this is:

```
~$ gcc -O3  test.c -o test
~$ ./test
a[0] - b = 9999999996
```

Now the result is correct. If we look at the assembly output again:

```
00000000004004f6 <main>:
  400404:       48 b8 00 e4 0b 54 02    movabs $0x2540be400,%rax
  40040b:       00 00 00
  40040e:       48 89 04 24             mov    %rax,(%rsp)
  400412:       48 c7 44 24 08 00 00    movq   $0x0,0x8(%rsp)
  400419:       00 00
  40041b:       48 c7 44 24 10 01 00    movq   $0x1,0x10(%rsp)
  400422:       00 00
  400424:       48 ff 04 24             incq   (%rsp)
  400428:       48 8b 04 24             mov    (%rsp),%rax
  400431:       48 8d 70 fb             lea    -0x5(%rax),%rsi
```

we will see one difference here which is in the last two lines:

```
  400428:       48 8b 04 24             mov    (%rsp),%rax
  400431:       48 8d 70 fb             lea    -0x5(%rax),%rsi
```

Instead of constant folding, `GCC` now preserves calculations in the assembly and places the value of `a[0]` in the `%rax` register afterwards. In the end it just subtracts the constant value of `b` from the `%rax` register and puts result to the `%rsi` .

Besides the `memory` specifier, we also see a new constraint here - `m` . This constraint tells the compiler to use the address of `a[0]` , instead of its value. So, now we are finished with `clobbers` and we may continue by looking at other constraints supported by `GCC` besides `r` and `m` which we have already seen.

Constraints

Now that we are finished with all three parts of an inline assembly statement, let's return to constraints. We already saw some constraints in the previous parts, like `r` which represents a `register` operand, `m` which represents a memory operand and `0-9` which represent an reused, indexed operand. Besides these `GCC` provides support for other constraints. For example the `i` constraint represents an `immediate` integer operand with know value:

```
#include <stdio.h>

int main(void)
{
        int a = 0;

        __asm__("movl %1, %0" : "=r"(a) : "i"(100));
        printf("a = %d\n", a);
        return 0;
}
```

The result is:

```
~$ gcc test.c -o test
~$ ./test
a = 100
```

Or for example `I` which represents an immediate 32-bit integer. The difference between `i` and `I` is that `i` is general, whereas `I` is strictly specified to 32-bit integer data. For example if you try to compile the following code:

```
unsigned long test_asm(int nr)
{
        unsigned long a = 0;

        __asm__("movq %1, %0" : "=r"(a) : "I"(0xffffffffff));
        return a;
}
```

you will get an error:

```
$ gcc -O3 test.c -o test
test.c: In function ‘test_asm’:
test.c:7:9: warning: asm operand 1 probably doesn’t match constraints
         __asm__("movq %1, %0" : "=r"(a) : "I"(0xffffffffff));
         ^
test.c:7:9: error: impossible constraint in ‘asm’
```

when at the same time:

```
unsigned long test_asm(int nr)
{
        unsigned long a = 0;

        __asm__("movq %1, %0" : "=r"(a) : "i"(0xffffffffffff));
        return a;
}
```

works perfectly:

```
~$ gcc -O3 test.c -o test
~$ echo $?
0
```

`GCC` also supports `J`, `K`, `N` constraints for integer constants in the range of 0-63 bits, signed 8-bit integer constants and unsigned 8-bit integer constants respectively. The `o` constraint represents a memory operand with an `offsetable` memory address. For example:

```
#include <stdio.h>

int main(void)
{
        static unsigned long arr[3] = {0, 1, 2};
        static unsigned long element;

        __asm__ volatile("movq 16+%1, %0" : "=r"(element) : "o"(arr));
        printf("%lu\n", element);
        return 0;
}
```

The result, as expected:

```
~$ gcc -O3 test.c -o test
~$ ./test
2
```

All of these constraints may be combined (so long as they do not conflict). In this case the compiler will choose the best one for a certain situation. For example:

```
#include <stdio.h>

unsigned long a = 1;

int main(void)
{
        unsigned long b;
        __asm__ ("movq %1,%0" : "=r"(b) : "r"(a));
        return b;
}
```

will use a memory operand:

```
0000000000400400 <main>:
  4004aa:       48 8b 05 6f 0b 20 00    mov    0x200b6f(%rip),%rax        # 601020 <a>
```

That's about all of the commonly used constraints in inline assembly statements. You can find more in the official documentation.

Architecture specific constraints

Before we finish, let's look at the set of special constraints. These constrains are architecture specific and as this book is specific to the x86_64 architecture, we will look at constraints related to it. First of all the set of `a` ... `d` and also `S` and `D` constraints represent generic purpose registers. In this case the `a` constraint corresponds to `%al` , `%ax` , `%eax` or `%rax` register depending on instruction size. The `S` and `D` constraints are `%si` and `%di` registers respectively. For example let's take our previous example. We can see in its assembly output that value of the `a` variable is stored in the `%eax` register. Now let's look at the assembly output of the same assembly, but with other constraint:

```
#include <stdio.h>

int a = 1;

int main(void)
{
        int b;
        __asm__ ("movq %1,%0" : "=r"(b) : "d"(a));
        return b;
}
```

Now we see that value of the `a` variable will be stored in the `%rax` register:

```
0000000000400400 <main>:
  4004aa:       48 8b 05 6f 0b 20 00    mov    0x200b6f(%rip),%rax        # 601020 <a>
```

The `f` and `t` constraints represent any floating point stack register - `%st` and the top of the floating point stack respectively. The `u` constraint represents the second value from the top of the floating point stack.

That's all. You may find more details about x86_64 and general constraints in the official documentation.

Links

- Linux kernel source code
- assembly programming language
- GCC
- GNU extension
- Global Descriptor Table
- Processor registers
- add instruction
- flags register
- x86_64
- constraints

杂项

这个章节包含不直接涉及到内核源码的部分以及各个子系统的实现。

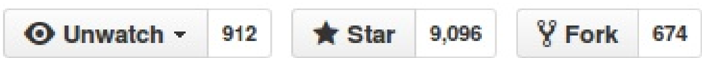
Linux 内核开发

简介

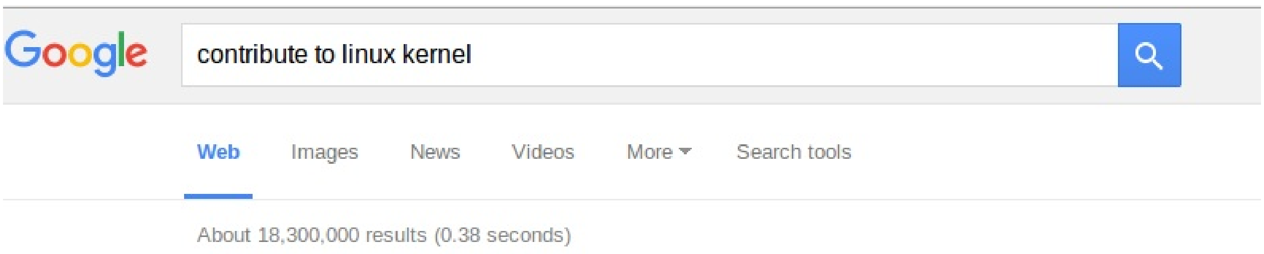
如你所知，我从去年开始写了一系列关于 `x86_64` 架构汇编语言程序设计的博文。除了大学期间写过一些 `Hello World` 这样无实用价值的程序之外，我从来没写过哪怕一行的底层代码。那些程序也是很久以前的事情了，就像我刚才说的，我几乎完全没有写过底层代码。直到不久前，我才开始对这些事情感兴趣，因为我意识到我虽然可以写出程序，但是我却不知道我的程序是怎样被组织运行的。

在写了一些汇编代码之后，我开始大致了解了程序在编译之后会变成什么样子。尽管如此，还是有很多其他的东西我不能够理解。例如：当 `syscall` 指令在我的汇编程序内执行时究竟发生了什么，当 `printf` 函数开始工作时又发生了什么，还有，我的程序是如何通过网络与其他计算机进行通信的。汇编语言并没有为这些问题带来答案，于是我决定做一番深入研究。我开始学习 Linux 内核的源代码，并且尝试着理解那些让我感兴趣的东西。然而 Linux 内核源代码也没有解答我所有的问题，不过我自身关于 Linux 内核及其外围流程的知识确实掌握的更好了。

在我开始学习 Linux 内核的九个半月之后，我写了这部分内容，并且发布了本书的第一部分。到现在为止，本书共包括了四个部分，而这并不是终点。我之所以写这一系列关于 Linux 内核的文章其实更多的是为了我自己。你也知道，Linux 内核的代码量极其巨大，另外还非常容易忘记这一块或那一块内核代码做了什么，或者忘记某些东西是怎么实现的。出乎意料的是 linux-insides 很快就火了，并且在九个月后积攒了 `9096` 个星星：



看起来人们对 Linux 内核的内在机制非常的感兴趣。除此之外，在我写 `linux-insides` 的这段时间里，我收到了很多人发来的问题，这些问题大都是关于如何开始向 Linux 内核贡献代码。通常来说，人们是很有兴趣为开源项目做贡献的，Linux 内核也不例外：



这么看起来大家对 Linux 内核的开发流程非常感兴趣。我认为如果这么一本关于 Linux 内核的书却不包括一部分来讲讲如何参与 Linux 内核开发的话，那就非常奇怪了。这就是我决定写这篇文章的原因。在本文中，你不会看到为什么你应该对贡献 Linux 内核感兴趣，但是如果你想参与 Linux 内核开发的话，那这部分就是为你而作。

让我们开始吧。

如何入门 Linux 内核

首先，让我们看看如何获取、构建并运行 Linux 内核。你可以通过两种方式来运行你自己定制的内核：

- 在虚拟机里运行 Linux 内核；
- 在真实的硬件上运行 Linux 内核。

我会对这两种方式都展开描述。在我们开始对 Linux 内核做些什么之前，我们首先需要先获取它。根据你目的的不同，有两种方式可以做到这一点。如果你只是想更新一下你电脑上的 Linux 内核版本，那么你可以使用特定于你 Linux 发行版的命令。

在这种情况下，你只需要使用软件包管理器下载新版本的 Linux 内核。例如，为了将 Ubuntu (Vivid Vervet) 系统的 Linux 内核更新至 `4.1` 版本，你只需要执行以下命令：

```
$ sudo add-apt-repository ppa:kernel-ppa/ppa
$ sudo apt-get update
```

在这之后，再执行下面的命令：

```
$ apt-cache showpkg linux-headers
```

然后选择你感兴趣的 Linux 内核的版本。最后，执行下面的命令并且将 `${version}` 替换为你从上一条命令的输出中选择的版本号。

```
$ sudo apt-get install linux-headers-${version} linux-headers-${version}-generic linux
-image-${version}-generic --fix-missing
```

最后重启你的系统。重启完成后，你将在 grub 菜单中看到新的内核。

另一方面，如果你对 Linux 内核开发感兴趣，那么你就需要获得 Linux 内核的源代码。你可以在 kernel.org 网站上找到它并且下载一个包含了 Linux 内核源代码的归档文件。实际上，Linux 内核的开发流程完全建立在 `git` 版本控制系统之上，所以你需要通过 `git` 来从 `kernel.org` 上获取内核源代码：

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

我不知道你怎么看，但是我本身是非常喜欢 `github` 的。它上面有一个 Linux 内核主线仓库的镜像，你可以通过以下命令克隆它：

```
$ git clone git@github.com:torvalds/linux.git
```

我是用我自己 fork 的仓库来进行开发的，等到我想从主线仓库拉取更新的时候，我只需要执行下方的命令即可：

```
$ git checkout master
$ git pull upstream master
```

注意这个主线仓库的远程主机名叫做 `upstream`。为了将主线 Linux 仓库添加为一个新的远程主机，你可以执行：

```
git remote add upstream git@github.com:torvalds/linux.git
```

在此之后，你将有两个远程主机：

```
~/dev/linux (master) $ git remote -v
origin    git@github.com:0xAX/linux.git (fetch)
origin    git@github.com:0xAX/linux.git (push)
upstream    https://github.com/torvalds/linux.git (fetch)
upstream    https://github.com/torvalds/linux.git (push)
```

其中一个远程主机是你的 fork 仓库 (`origin`)，另一个是主线仓库 (`upstream`)。

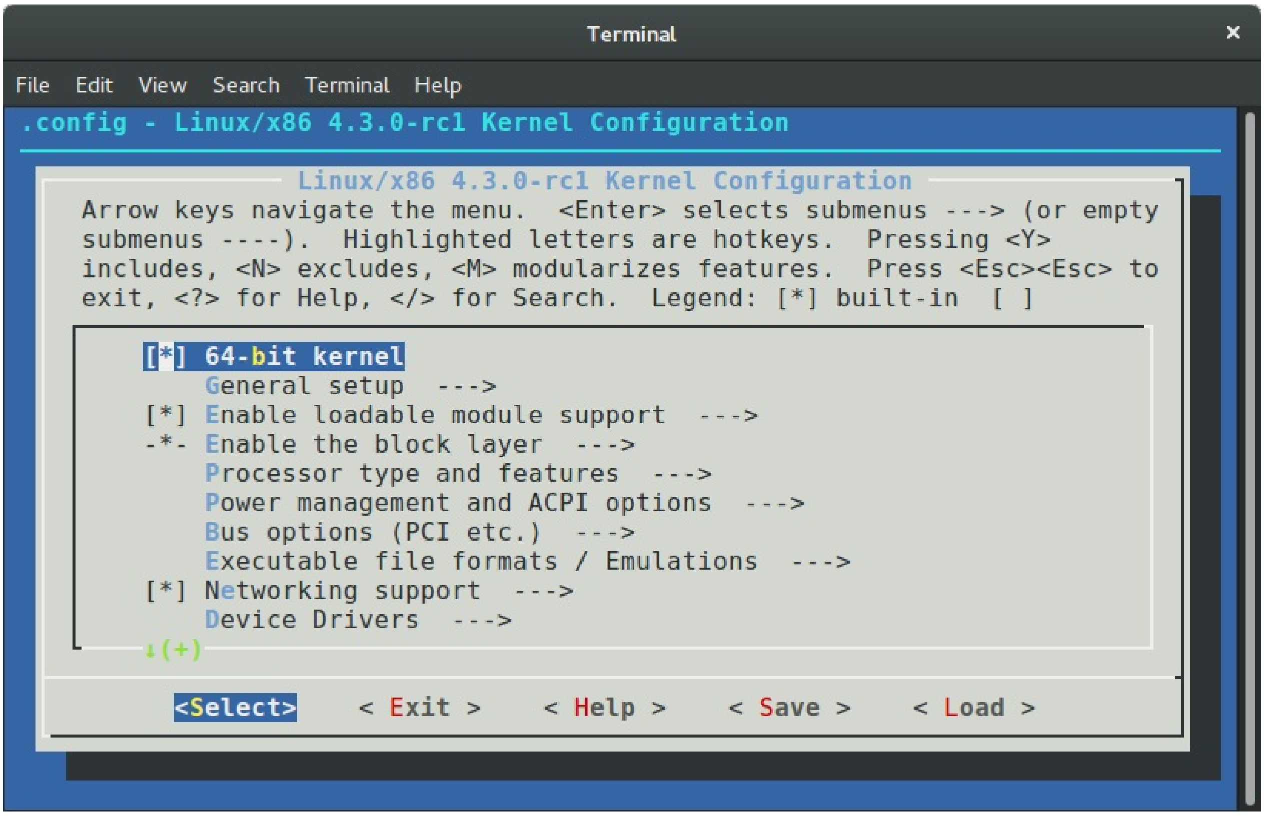
现在，我们已经有了一份 Linux 内核源代码的本地副本，我们需要配置并编译内核。Linux 内核的配置有很多不同的方式，最简单的方式就是直接拷贝 `/boot` 目录下已安装内核的配置文件：

```
$ sudo cp /boot/config-$(uname -r) ~/dev/linux/.config
```

如果你当前的内核被编译为支持访问 `/proc/config.gz` 文件，你也可以使用以下命令复制当前内核的配置文件：

```
$ cat /proc/config.gz | gunzip > ~/dev/linux/.config
```

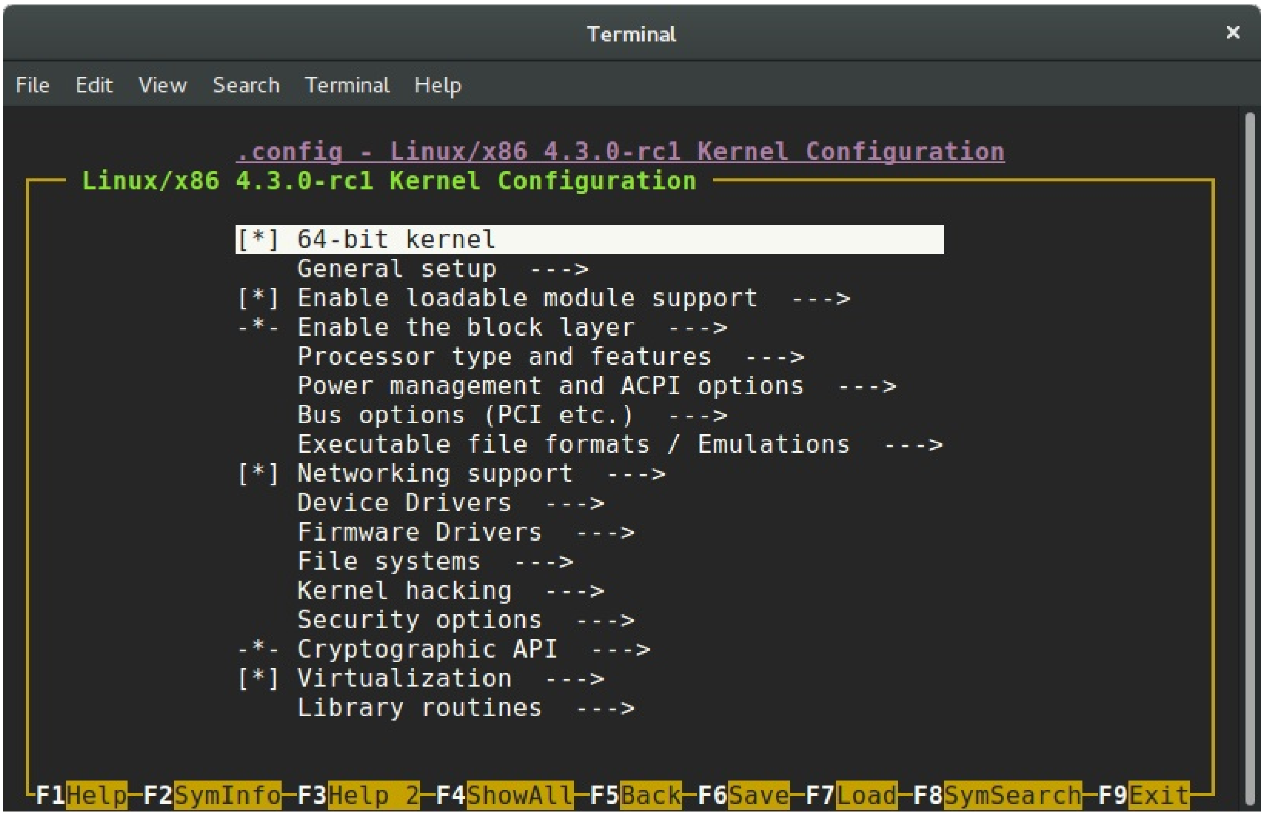
如果你对发行版维护者提供的标准内核配置文件并不满意，你也可以手动配置 Linux 内核，有两种方式可以做到这一点。Linux 内核的根 Makefile 文件提供了一系列可配置的目标选项。例如 `menuconfig` 为内核配置提供了一个菜单界面：



`defconfig` 参数会为当前的架构生成默认的内核配置文件，例如 x86_64 defconfig。你可以将 `ARCH` 命令行参数传递给 `make`，以此来为给定架构创建 `defconfig` 配置文件：

```
$ make ARCH=arm64 defconfig
```

`allnoconfig`、`allyesconfig` 以及 `allmodconfig` 参数也允许你生成新的配置文件，其效果分别为尽可能多的选项都关闭、尽可能多的选项都启用或尽可能多的选项都作为模块启用。`nconfig` 命令行参数提供了基于 `ncurses` 的菜单程序来配置 Linux 内核：



`randconfig` 参数甚至可以随机地生成 Linux 内核配置文件。我不会讨论如何去配置 Linux 内核或启用哪个选项，因为没有必要这么做：首先，我不知道你的硬件配置；其次，如果我知道了你的硬件配置，那么剩下的问题就是搞清楚如何使用程序生成内核配置，而这些程序的使用都是非常容易的。

好了，我们现在有了 Linux 内核的源代码并且完成了配置。下一步就是编译 Linux 内核了。最简单的编译 Linux 内核的方式就是执行以下命令：

```
$ make
scripts/kconfig/conf  --silentoldconfig Kconfig
#
# configuration written to .config
#
  CHK     include/config/kernel.release
  UPD     include/config/kernel.release
  CHK     include/generated/uapi/linux/version.h
  CHK     include/generated/utsrelease.h
  ...
  ...
  ...
  OBJCOPY arch/x86/boot/vmlinux.bin
  AS      arch/x86/boot/header.o
  LD      arch/x86/boot/setup.elf
  OBJCOPY arch/x86/boot/setup.bin
  BUILD   arch/x86/boot/bzImage
  Setup is 15740 bytes (padded to 15872 bytes).
System is 4342 kB
CRC 82703414
Kernel: arch/x86/boot/bzImage is ready  (#73)
```

为了增加内核的编译速度，你可以给 `make` 传递命令行参数 `-jN` ，这里的 `N` 指定了并发执行的命令数目：

```
$ make -j8
```

如果你想为一个架构构建一个与当前内核不同的内核，那么最简单的方式就是传递下面两个参数：

- `ARCH` 命令行参数是目标架构名；
- `CROSS_COMPILER` 命令行参数是交叉编译工具的前缀；

例如，如果我们想使用默认内核配置文件为 arm64 架构编译 Linux 内核，我们需要执行以下命令：

```
$ make -j4 ARCH=arm64 CROSS_COMPILER=aarch64-linux-gnu- defconfig
$ make -j4 ARCH=arm64 CROSS_COMPILER=aarch64-linux-gnu-
```

编译的结果就是你会看到压缩后的内核文件 - `arch/x86/boot/bzImage` 。既然我们已经编译好了内核，那么就可以把它安装到我们的电脑上或者只是将它运行在模拟器里。

安装 Linux 内核

就像我之前写的，我们将考察两种运行新内核的方法：第一种情况，我们可以在真实的硬件上安装并运行新版本的 Linux 内核，第二种情况就是在虚拟机上运行 Linux 内核。在前面的段落中我们看到了如何从源代码来构建 Linux 内核，并且我们现在已经得到了内核的压缩镜像：

```
...
...
...
Kernel: arch/x86/boot/bzImage is ready  (#73)
```

在我们获得了 bzImage 之后，我们需要使用以下命令来为新的 Linux 内核安装 `headers` 和 `modules` ：

```
$ sudo make headers_install
$ sudo make modules_install
```

以及内核自身：

```
$ sudo make install
```

从这时起，我们已经安装好了新版本的 Linux 内核，现在我们需要通知 `bootloader` 新内核已经安装完成。我们当然可以手动编辑 `/boot/grub2/grub.cfg` 配置文件并将新内核添加进去，但是我更推荐使用脚本来完成这件事。我现在在使用两种不同的 Linux 发行版：Fedora 和 Ubuntu，有两种方式可以用来更新 grub 配置文件，我目前正在使用下面的脚本来达到这一目的：

```
#!/bin/bash

source "term-colors"

DISTRIBUTIVE=$(cat /etc/*-release | grep NAME | head -1 | sed -n -e 's/NAME\=//p')
echo -e "Distributive: ${Green}${DISTRIBUTIVE}${Color_Off}"

if [[ "$DISTRIBUTIVE" == "Fedora" ]] ;
then
    su -c 'grub2-mkconfig -o /boot/grub2/grub.cfg'
else
    sudo update-grub
fi

echo "${Green}Done.${Color_Off}"
```

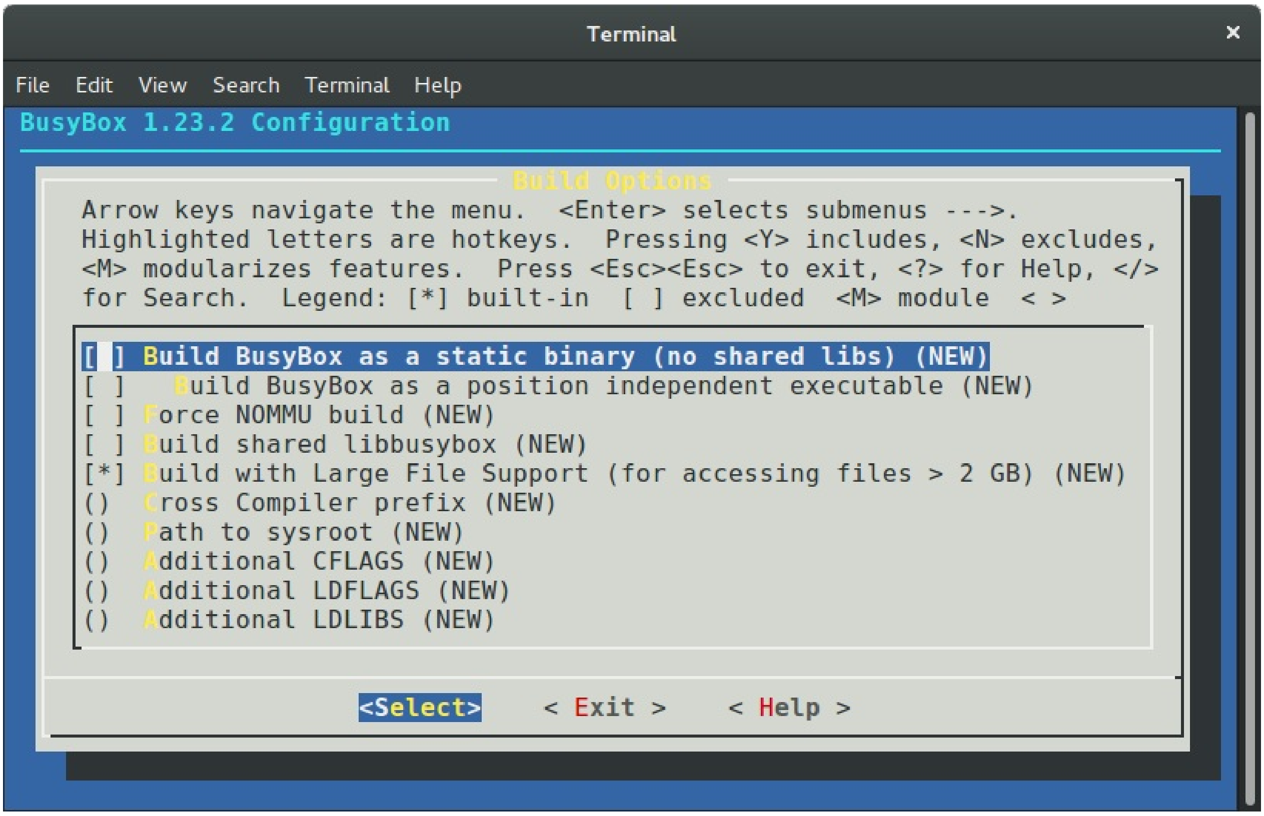
这是新 Linux 内核安装过程中的最后一步，在这之后你可以重启你的电脑，然后在启动过程中选择新版本的内核。

第二种情况就是在虚拟机内运行新的 Linux 内核，我更倾向于使用 qemu。首先我们需要为此构建初始的虚拟内存盘 - initrd。 `initrd` 是一个临时的根文件系统，它在初始化期间被 Linux 内核使用，而那时其他的文件系统尚未被挂载。我们可以使用以下命令构建 `initrd` ：

首先我们需要下载 busybox，然后运行 `menuconfig` 命令配置它：

```
$ mkdir initrd
$ cd initrd
$ curl http://busybox.net/downloads/busybox-1.23.2.tar.bz2 | tar xjf -
$ cd busybox-1.23.2/
$ make menuconfig
$ make -j4
```

`busybox` 是一个可执行文件 - `/bin/busybox` ，它包括了一系列类似于 coreutils 的标准工具。在 `busysbox` 菜单界面上我们需要启用 `Build BusyBox as a static binary (no shared libs)` 选项：



我们可以按照下方的路径找到这个菜单项：

```
Busybox Settings
--> Build Options
```

之后，我们从 `busysbox` 的配置菜单退出去，然后执行下面的命令来构建并安装它：

```
$ make -j4
$ sudo make install
```

既然 `busybox` 已经安装完了，那么我们就可以开始构建 `initrd` 了。为了完成构建过程，我们需要返回到之前的 `initrd` 目录并且运行命令：

```
$ cd ..
$ mkdir -p initramfs
$ cd initramfs
$ mkdir -pv {bin,sbin,etc,proc,sys,usr/{bin,sbin}}
$ cp -av ../busybox-1.23.2/_install/* .
```

这会把 `busybox` 复制到 `bin` 目录、`sbin` 目录以及其他相关目录内。现在我们需要创建可执行的 `init` 文件，该文件将会在系统内作为第一个进程执行。我的 `init` 文件仅仅挂载了 procfs 和 sysfs 文件系统并且执行了 shell 程序：

```
#!/bin/sh

mount -t proc none /proc
mount -t sysfs none /sys

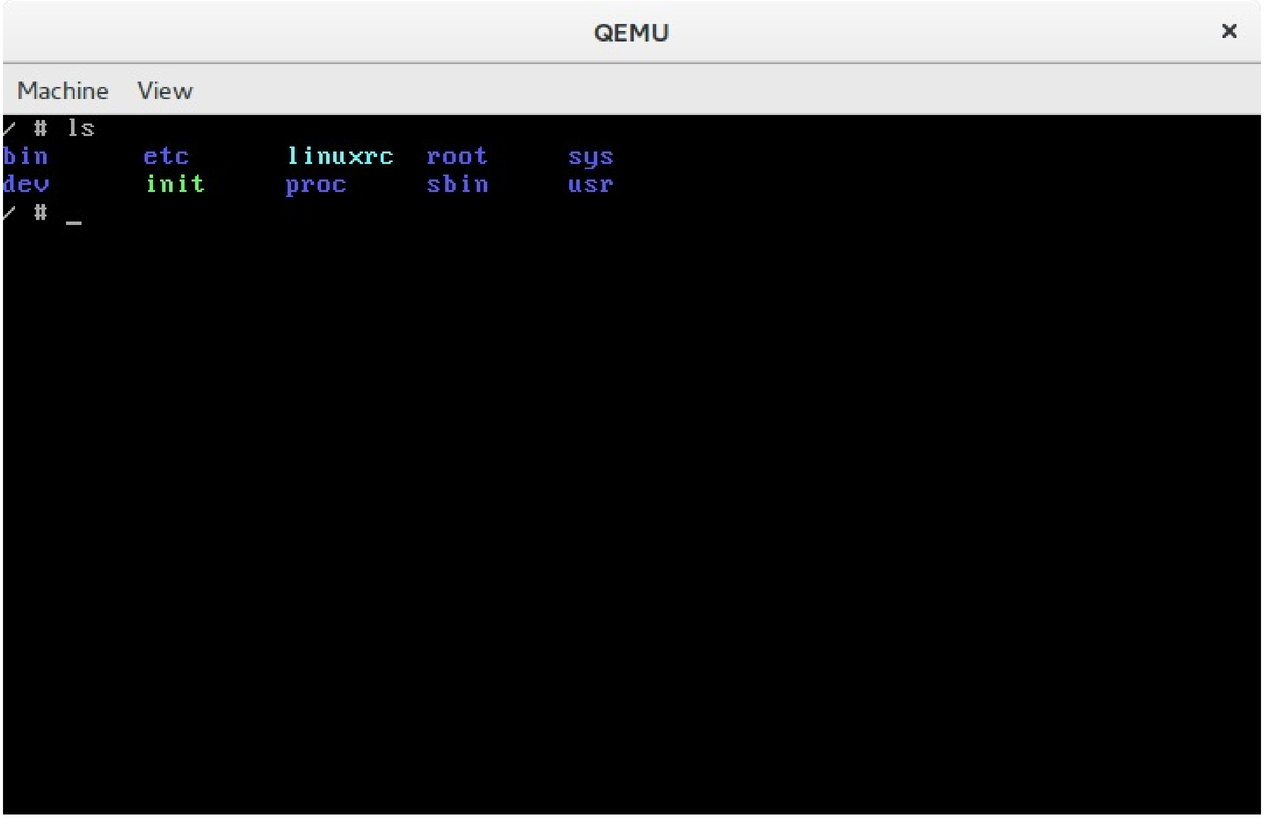
exec /bin/sh
```

最后，我们创建一个归档文件，这就是我们的 `initrd` 了：

```
$ find . -print0 | cpio --null -ov --format=newc | gzip -9 > ~/dev/initrd_x86_64.gz
```

我们现在可以在虚拟机里运行内核了。就像我之前写过的，我偏向于使用 qemu 来完成这些工作，下面的命令可以用来运行我们的 Linux 内核：

```
$ qemu-system-x86_64 -snapshot -m 8GB -serial stdio -kernel ~/dev/linux/arch/x86_64/bo
ot/bzImage -initrd ~/dev/initrd_x86_64.gz -append "root=/dev/sda1 ignore_loglevel"
```



从现在起，我们就可以在虚拟机内运行 Linux 内核了，这意味着我们可以开始对内核进行修改和测试了。

除了上面的手动过程之外，还可以考虑使用 ivandaviov/minimal 来自动生成 `initrd` 。

Linux 内核开发入门

这部分的核心内容主要回答了两个问题：在你发送第一个 Linux 内核补丁之前你应该做什么 (`to do`) 和不能做什么 (`not to do`)。请千万不要把应该做的事 (`to do`) 和待办事项 (`todo`) 搞混了。我无法回答你能为 Linux 内核修复什么问题，我只是想告诉你我拿 Linux 内核源代码做实验的过程。

首先，我需要使用以下命令从 Linus 的仓库中拉取最新的更新：

```
$ git checkout master
$ git pull upstream master
```

在这之后，我的本地 Linux 内核源代码仓库已经和主线仓库同步了。现在我们可以在源代码上做些修改了。就像我之前写的，关于从哪开始修改或者可以做些什么，我并不能给你太多建议。不过，对于新手来说最好的地方就是 `staging` 源码树，也就是 drivers/staging 上的驱动集合。 `staging` 源码树的主要维护者是 Greg Kroah-Hartman，该源码树正是你的琐碎补丁可以被接受的地方。让我们看一个简单的例子，该例子描述了如何生成补丁、检查补丁以及如何将补丁发送到 Linux 内核邮件列表。

如果我们查看一下为 Digi International EPCA PCI 基础设备所写的驱动程序，在 295 行我们将会看到 `dgap_sindex` 函数：

```
static char *dgap_sindex(char *string, char *group)
{
    char *ptr;

    if (!string || !group)
        return NULL;

    for (; *string; string++) {
        for (ptr = group; *ptr; ptr++) {
            if (*ptr == *string)
                return string;
        }
    }

    return NULL;
}
```

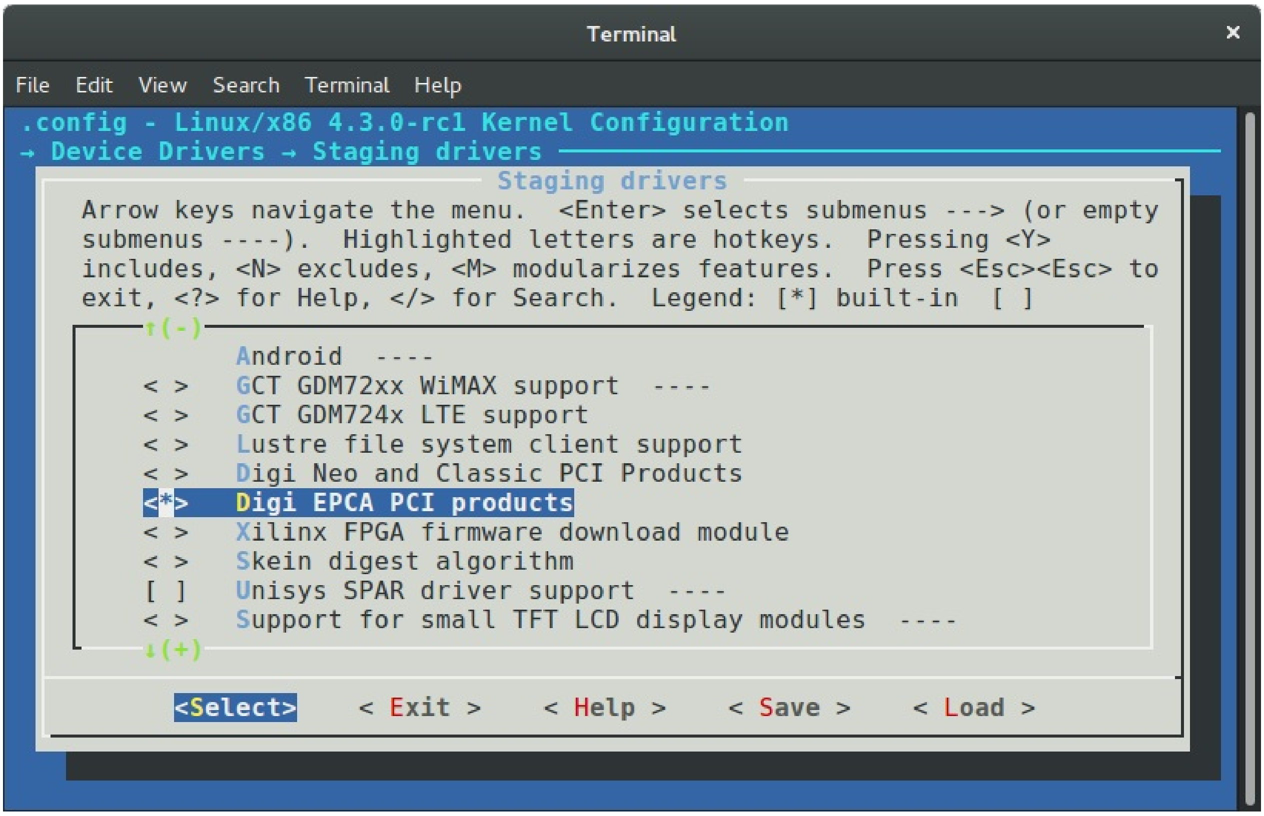
这个函数查找 `group` 和 `string` 共有的字符并返回其位置。在研究 Linux 内核源代码期间，我注意到 lib/string.c 文件里实现了一个 `strpbrk` 函数，该函数和 `dgap_sinidex` 函数做了同样的事。使用现存函数的另一种自定义实现并不是一个好主意，所以我们可以从 drivers/staging/dgap/dgap.c 源码文件中移除 `dgap_sindex` 函数并使用 `strpbrk` 替换它。

首先，让我们基于当前主分支创建一个新的 `git` 分支，该分支与 Linux 内核主仓库同步：

```
$ git checkout -b "dgap-remove-dgap_sindex"
```

然后，我们可以将 `dgap_sindex` 函数替换为 `strpbrk`。做完这些修改之后，我们需要重新编译 Linux 内核或者只重编译 dgap 目录。不要忘了在内核配置文件中启用这个驱动，你可以在如下位置找到该驱动：

```
Device Drivers
--> Staging drivers
----> Digi EPCA PCI products
```



现在是时候提交修改了，我使用下面的命令组合来完成这件事：

```
$ git add .
$ git commit -s -v
```

最后一条命令运行后将会打开一个编辑器，该编辑器会从 `$GIT_EDITOR` 或 `$EDITOR` 环境变量中进行选择。 `-s` 命令行参数会在提交信息的末尾按照提交者名字加上一行 `Signed-off-by` 。你在每一条提交信息的最后都能看到这一行，例如 - 00cc1633。这一行的主要目的是追踪谁做的修改。 `-v` 选项按照合并格式显示 `HEAD` 提交和即将进行的最新提交之间的差异。这样做不是并必须的，但有时候却很有用。再来说下提交信息，实际上，一条提交信息由两部分组成：

第一部分放在第一行，它包括了一句对所做修改的简短描述。这一行以 `[PATCH]` 做前缀，后面跟上子系统、驱动或架构的名字，以及在 `:` 之后的简述信息。在我们这个例子中，这一行信息如下所示：

```
[PATCH] staging/dgap: Use strpbrk() instead of dgap_sindex()
```

在简述信息之后，我们通常空一行再加上对本次提交的详尽描述。在我们的这个例子中，这些信息如下所示：

```
The <linux/string.h> provides strpbrk() function that does the same that the
dgap_sindex(). Let's use already defined function instead of writing custom.
```

在提交信息的最后是 `Sign-off-by` 这一行。注意，提交信息的每一行不能超过 `80` 个字符并且提交信息必须详细地描述你所做的修改。千万不要只写一条类似于 `Custom function removed` 这样的信息，你需要描述你做了什么以及为什么这样做。补丁的审核者必须据此知道他们正在审核什么内容，除此之外，这里的提交信息本身也非常有用。每当你不能理解一些东西的时候，我们都可以使用 git blame 命令来阅读关于修改的描述。

提交修改之后，是时候生成补丁文件了。我们可以使用 `format-patch` 命令来完成：

```
$ git format-patch master
0001-staging-dgap-Use-strpbrk-instead-of-dgap_sindex.patch
```

我们把分支名字 (这里是 `master`) 传递给 `format-patch` 命令，该命令会根据那些包括在 `dgap-remove-dgap_sindex` 分支但不在 `master` 分支的最新改动来生成补丁。你会发现，`format-patch` 命令生成的文件包含了最新所做的修改，该文件的名字是基于提交信息的简述来生成的。如果你想按照自定义的文件名来生成补丁，你可以使用 `--stdout` 选项：

```
$ git format-patch master --stdout > dgap-patch-1.patch
```

最后一步就是在我们生成补丁之后将之发送到 Linux 内核邮件列表。当然，你可以使用任意的邮件客户端，不过 `git` 为此提供了一个专门的命令：`git send-email` 。在发送补丁之前，你需要知道发到哪里。虽然你可以直接把它发送到 `linux-kernel@vger.kernel.org` 这个邮件列表，但这很可能让你的补丁因为巨大的消息流而被忽略掉。最好的选择是将补丁发送到你的修改所属子系统的维护者那里。你可以使用 `get_maintainer.pl` 这个脚本来找到这些维护者的名字。你所需要做的就是将你代码所在的文件或目录作为参数传递给脚本。

```
$ ./scripts/get_maintainer.pl -f drivers/staging/dgap/dgap.c
Lidza Louina <lidza.louina@gmail.com> (maintainer:DIGI EPCA PCI PRODUCTS)
Mark Hounschell <markh@compro.net> (maintainer:DIGI EPCA PCI PRODUCTS)
Daeseok Youn <daeseok.youn@gmail.com> (maintainer:DIGI EPCA PCI PRODUCTS)
Greg Kroah-Hartman <gregkh@linuxfoundation.org> (supporter:STAGING SUBSYSTEM)
driverdev-devel@linuxdriverproject.org (open list:DIGI EPCA PCI PRODUCTS)
devel@driverdev.osuosl.org (open list:STAGING SUBSYSTEM)
linux-kernel@vger.kernel.org (open list)
```

你将会看到一组姓名和与之相关的邮件地址。现在你可以通过下面的命令发送补丁了：

```
$ git send-email --to "Lidza Louina <lidza.louina@gmail.com>" \
  --cc "Mark Hounschell <markh@compro.net>"                   \
  --cc "Daeseok Youn <daeseok.youn@gmail.com>"                \
  --cc "Greg Kroah-Hartman <gregkh@linuxfoundation.org>"      \
  --cc "driverdev-devel@linuxdriverproject.org"               \
  --cc "devel@driverdev.osuosl.org"                           \
  --cc "linux-kernel@vger.kernel.org"
```

这就是全部的过程。补丁被发出去了，现在你所需要做的就是等待 Linux 内核开发者的反馈。在你发送完补丁并且维护者接受它之后，你将在维护者的仓库中看到它 (例如前文你看到的补丁)。一段时间后，维护者将会向 Linus 发送一个拉取请求，之后你就会在主线仓库里看到你的补丁了。

这就是全部内容。

一些建议

在该部分的最后，我想给你一些建议，这些建议大都是关于在 Linux 内核的开发过程中需要做什么以及不能做什么的：

- 考虑，考虑，再考虑。在你决定发送补丁之前再三考虑。
- 在你每次改完 Linux 内核源代码之后 - 试着编译它。我指的是任何修改之后，都要不断的编译。没有人喜欢那些连编译都不通过修改。
- Linux 内核有一套代码规范指南，你需要遵守它。有一个很棒的脚本可以帮你检查所做的修改。这个脚本就是 - scripts/checkpatch.pl。只需要将被改动的源码文件传递给它即可，然后你就会看到如下输出：

```
$ ./scripts/checkpatch.pl -f drivers/staging/dgap/dgap.c
WARNING: Block comments use * on subsequent lines
#94: FILE: drivers/staging/dgap/dgap.c:94:
+/*
+     SUPPORTED PRODUCTS

CHECK: spaces preferred around that '|' (ctx:VxV)
#143: FILE: drivers/staging/dgap/dgap.c:143:
+    { PPCM,        PCI_DEV_XEM_NAME,     64, (T_PCXM|T_PCLITE|T_PCIBUS) },
```

在 `git diff` 命令的帮助下，你也会看到一些有问题的地方：

```
~/dev/linux (dgap-remove-dgap_sindex) $ git diff
diff --git a/init/main.c b/init/main.c
index 9e64d70..af379a5 100644
--- a/init/main.c
+++ b/init/main.c
@@ -153,6 +153,8 @@ EXPORT_SYMBOL(reset_devices);
 static int __init set_reset_devices(char *str)
 {
        reset_devices = 1;
+
+
        return 1;
 }
```

* Linus 不接受 github pull requests

* 如果你的修改是由一些不同的且不相关的改动所组成的，你需要通过分离提交来切分修改。`git format-patch` 命令将会为每个提交生成一个补丁，每个补丁的标题会包含一个 `vN` 前缀，其中 `N` 是补丁的编号。如果你打算发送一系列补丁，也许给 `git format-patch` 命令传递 `--cover-letter` 选项会对此很有帮助。这会生成一个附加文件，该文件包括的附函可以用来描述你的补丁集所做的改动。在 `git send-email` 命令中使用 `--in-reply-to` 选项也是一个好主意，该选项允许你将补丁集作为对附函的回复发送出去。对于维护者来说，你补丁集的结构看起来就像下面这样：

```
|--> cover letter
  |----> patch_1
  |----> patch_2
```

你可以将 `message-id` 参数传递给 `--in-reply-to` 选项，该选项可以在 `git send-email` 命令的输出中找到。

有一件非常重要的事，那就是你的邮件必须是纯文本格式。通常来说，`send-email` 和 `format-patch` 这两个命令在内核开发中都是非常有用的，所以请查阅这些命令的的相关文档，你会发现很多有用的选项，例如：git send-email 和 git format-patch。

* 如果你发完补丁之后没有得到立即答复，请不要惊讶，因为维护者们都是很忙的。

* scripts 目录包含了很多对 Linux 内核开发有用的脚本。我们已经看过此目录中的两个脚本了：`checkpatch.pl` 和 `get_maintainer.pl`。除此之外，你还可以找到 stackusage 脚本，它可以打印栈的使用情况，extract-vmlinux 脚本可以提取出未经压缩的内镜镜像，还有很多其他的脚本。在 `scripts` 目录之外，你也会发现很多有用的脚本，这些脚本是 Lorenzo Stoakes 为内核开发而编写的。

* 订阅 Linux 内核邮件列表。`lkml` 列表中每天都会有大量的信件，但是阅读它们并了解一些类似于 Linux 内核目前开发状态的内容是很有帮助的。除了 `lkml` 之外，还有一些其他的邮件列表，它们分别对应于不同的 Linux 内核子系统。

* 如果你发的补丁第一次没有被接受，你就会收到 Linux 内核开发者的反馈。请做一些修改然后以 `[PATCH vN]`（`N` 是补丁版本号）为前缀重新发送补丁，例如：

```
[PATCH v2] staging/dgap: Use strpbrk() instead of dgap_sindex()
```

同样的，这次的补丁也必须包括更新日志以便描述自上一次的补丁以来所做的修改。当然，本文并不是对 Linux 内核开发详尽无遗的指导清单，但是一些最重要的事项已经都被阐明了。

Happy Hacking!

总结

我希望这篇文章能够帮助其他人加入 Linux 内核社区！ 如果你有其他问题或建议，可以给我写邮件或者在 Twitter 上联系我。

请注意，英语并不是我的母语，对此带来的不便我感到很抱歉。如果你发现了错误，请通过邮件或发 PR 来通知我。

相关链接

- blog posts about assembly programming for x86_64
- Assembler
- distro
- package manager
- grub
- kernel.org
- version control system
- arm64
- bzImage
- qemu
- initrd
- busybox
- coreutils
- procfs
- sysfs
- Linux kernel mail listing archive
- Linux kernel coding style guide
- How to Get Your Change Into the Linux Kernel
- Linux Kernel Newbies
- plain text

你知道 Linux 内核是如何构建的吗？

介绍

我不会告诉你怎么在自己的电脑上去构建、安装一个定制化的 Linux 内核，这样的资料太多了，它们会对你有帮助。本文会告诉你当你在内核源码路径里敲下 `make` 时会发生什么。

当我刚刚开始学习内核代码时，Makefile 是我打开的第一个文件，这个文件看起来真令人害怕 :)。那时候这个 Makefile 还只包含了 `1591` 行代码，当我开始写本文时，内核已经是 4.2.0 的第三个候选版本 了。

这个 Makefile 是 Linux 内核代码的根 Makefile ，内核构建就始于此处。是的，它的内容很多，但是如果你已经读过内核源代码，你就会发现每个包含代码的目录都有一个自己的 Makefile 。当然了，我们不会去描述每个代码文件是怎么编译链接的，所以我们将只会挑选一些通用的例子来说明问题。而你不会在这里找到构建内核的文档、如何整洁内核代码、tags 的生成和交叉编译 相关的说明，等等。我们将从 `make` 开始，使用标准的内核配置文件，到生成了内核镜像 bzImage 结束。

如果你已经很了解 make 工具，那是最好，但是我也会描述本文出现的相关代码。

让我们开始吧！

编译内核前的准备

在开始编译前要进行很多准备工作。最主要的就是找到并配置好配置文件，`make` 命令要使用到的参数都需要从这些配置文件获取。现在就让我们深入内核的根 `Makefile` 吧。

内核的根 `Makefile` 负责构建两个主要的文件：vmlinux （内核镜像可执行文件）和模块文件。内核的 Makefile 从定义如下变量开始：

```
VERSION = 4
PATCHLEVEL = 2
SUBLEVEL = 0
EXTRAVERSION = -rc3
NAME = Hurr durr I'ma sheep
```

这些变量决定了当前内核的版本，并且被使用在很多不同的地方，比如同一个 `Makefile` 中的 `KERNELVERSION` ：

```
KERNELVERSION = $(VERSION)$(if $(PATCHLEVEL),.$(PATCHLEVEL)$(if $(SUBLEVEL),.$(SUBLEVE
L)))$(EXTRAVERSION)
```

接下来我们会看到很多 `ifeq` 条件判断语句，它们负责检查传递给 `make` 的参数。内核的 `Makefile` 提供了一个特殊的编译选项 `make help` ，这个选项可以生成所有的可用目标和一些能传给 `make` 的有效的命令行参数。举个例子， `make V=1` 会在构建过程中输出详细的编译信息，第一个 `ifeq` 就是检查传递给 `make` 的 `V=n` 选项。

```
ifeq ("$(origin V)", "command line")
  KBUILD_VERBOSE = $(V)
endif
ifndef KBUILD_VERBOSE
  KBUILD_VERBOSE = 0
endif

ifeq ($(KBUILD_VERBOSE),1)
  quiet =
  Q =
else
  quiet=quiet_
  Q = @
endif

export quiet Q KBUILD_VERBOSE
```

如果 `V=n` 这个选项传给了 `make` ，系统就会给变量 `KBUILD_VERBOSE` 选项附上 `V` 的值，否则的话， `KBUILD_VERBOSE` 就会为 `0` 。然后系统会检查 `KBUILD_VERBOSE` 的值，以此来决定 `quiet` 和 `Q` 的值。符号 `@` 控制命令的输出，如果它被放在一个命令之前，这条命令的输出将会是 `CC scripts/mod/empty.o` ，而不是 `Compiling .... scripts/mod/empty.o` （LCTT 译注：CC 在 Makefile 中一般都是编译命令）。在这段最后，系统导出了所有的变量。

下一个 `ifeq` 语句检查的是传递给 `make` 的选项 `O=/dir`，这个选项允许在指定的目录 `dir` 输出所有的结果文件：

```
ifeq ($(KBUILD_SRC),)

ifeq ("$(origin O)", "command line")
  KBUILD_OUTPUT := $(O)
endif

ifneq ($(KBUILD_OUTPUT),)
saved-output := $(KBUILD_OUTPUT)
KBUILD_OUTPUT := $(shell mkdir -p $(KBUILD_OUTPUT) && cd $(KBUILD_OUTPUT) \
                                && /bin/pwd)
$(if $(KBUILD_OUTPUT),, \
     $(error failed to create output directory "$(saved-output)"))

sub-make: FORCE
    $(Q)$(MAKE) -C $(KBUILD_OUTPUT) KBUILD_SRC=$(CURDIR) \
    -f $(CURDIR)/Makefile $(filter-out _all sub-make,$(MAKECMDGOALS))

skip-makefile := 1
endif # ifneq ($(KBUILD_OUTPUT),)
endif # ifeq ($(KBUILD_SRC),)
```

系统会检查变量 `KBUILD_SRC` ，它代表内核代码的顶层目录，如果它是空的（第一次执行 makefile 时总是空的），我们会设置变量 `KBUILD_OUTPUT` 为传递给选项 `O` 的值（如果这个选项被传进来了）。下一步会检查变量 `KBUILD_OUTPUT` ，如果已经设置好，那么接下来会做以下几件事：

- 将变量 `KBUILD_OUTPUT` 的值保存到临时变量 `saved-output` ；
- 尝试创建给定的输出目录；
- 检查创建的输出目录，如果失败了就打印错误；
- 如果成功创建了输出目录，那么就在新目录重新执行 `make` 命令（参见选项 `-C` ）。

下一个 `ifeq` 语句会检查传递给 make 的选项 `C` 和 `M` ：

```
ifeq ("$(origin C)", "command line")
  KBUILD_CHECKSRC = $(C)
endif
ifndef KBUILD_CHECKSRC
  KBUILD_CHECKSRC = 0
endif

ifeq ("$(origin M)", "command line")
  KBUILD_EXTMOD := $(M)
endif
```

第一个选项 `C` 会告诉 `Makefile` 需要使用环境变量 `$CHECK` 提供的工具来检查全部 `c` 代码，默认情况下会使用 sparse 。第二个选项 `M` 会用来编译外部模块（本文不做讨论）。

系统还会检查变量 `KBUILD_SRC` ，如果 `KBUILD_SRC` 没有被设置，系统会设置变量 `srctree` 为 `.` ：

```
ifeq ($(KBUILD_SRC),)
        srctree := .
endif

objtree    := .
src        := $(srctree)
obj        := $(objtree)

export srctree objtree VPATH
```

这将会告诉 `Makefile` 内核的源码树就在执行 `make` 命令的目录，然后要设置 `objtree` 和其他变量为这个目录，并且将这些变量导出。接着就是要获取 `SUBARCH` 的值，这个变量代表了当前的系统架构（LCTT 译注：一般都指CPU 架构）：

```
SUBARCH := $(shell uname -m | sed -e s/i.86/x86/ -e s/x86_64/x86/ \
                  -e s/sun4u/sparc64/ \
                  -e s/arm.*/arm/ -e s/sa110/arm/ \
                  -e s/s390x/s390/ -e s/parisc64/parisc/ \
                  -e s/ppc.*/powerpc/ -e s/mips.*/mips/ \
                  -e s/sh[234].*/sh/ -e s/aarch64.*/arm64/ )
```

如你所见，系统执行 uname 得到机器、操作系统和架构的信息。因为我们得到的是 `uname` 的输出，所以我们需要做一些处理再赋给变量 `SUBARCH` 。获得 `SUBARCH` 之后就要设置 `SRCARCH` 和 `hfr-arch` ，`SRCARCH` 提供了硬件架构相关代码的目录，`hfr-arch` 提供了相关头文件的目录：

```
ifeq ($(ARCH),i386)
        SRCARCH := x86
endif
ifeq ($(ARCH),x86_64)
        SRCARCH := x86
endif

hdr-arch  := $(SRCARCH)
```

注意： `ARCH` 是 `SUBARCH` 的别名。如果没有设置过代表内核配置文件路径的变量 `KCONFIG_CONFIG` ，下一步系统会设置它，默认情况下就是 `.config` ：

```
KCONFIG_CONFIG    ?= .config
export KCONFIG_CONFIG
```

以及编译内核过程中要用到的 shell

```
CONFIG_SHELL := $(shell if [ -x "$$BASH" ]; then echo $$BASH; \
      else if [ -x /bin/bash ]; then echo /bin/bash; \
      else echo sh; fi ; fi)
```

接下来就要设置一组和编译内核的编译器相关的变量。我们会设置主机的 `c` 和 `C++` 的编译器及相关配置项：

```
HOSTCC       = gcc
HOSTCXX      = g++
HOSTCFLAGS   = -Wall -Wmissing-prototypes -Wstrict-prototypes -O2 -fomit-frame-pointer
 -std=gnu89
HOSTCXXFLAGS = -O2
```

接下来会去适配代表编译器的变量 `CC`，那为什么还要 `HOST*` 这些变量呢？这是因为 `CC` 是编译内核过程中要使用的目标架构的编译器，但是 `HOSTCC` 是要被用来编译一组 `host` 程序的（下面我们就会看到）。

然后我们就看到变量 `KBUILD_MODULES` 和 `KBUILD_BUILTIN` 的定义，这两个变量决定了我们要编译什么东西（内核、模块或者两者）：

```
KBUILD_MODULES :=
KBUILD_BUILTIN := 1

ifeq ($(MAKECMDGOALS),modules)
  KBUILD_BUILTIN := $(if $(CONFIG_MODVERSIONS),1)
endif
```

在这我们可以看到这些变量的定义，并且，如果们仅仅传递了 `modules` 给 `make`，变量 `KBUILD_BUILTIN` 会依赖于内核配置选项 `CONFIG_MODVERSIONS` 。

下一步操作是引入下面的文件：

```
include scripts/Kbuild.include
```

文件 Kbuild 或者又叫做 `Kernel Build System` 是一个用来管理构建内核及其模块的特殊框架。`Kbuild` 文件的语法与 Makefile 一样。文件 scripts/Kbuild.include 为 `Kbuild` 系统提供了一些常规的定义。因为我们包含了这个 `Kbuild` 文件，我们可以看到和不同工具关联的这些变量的定义，这些工具会在内核和模块编译过程中被使用（比如链接器、编译器、来自 binutils 的二进制工具包）：

```
AS          = $(CROSS_COMPILE)as
LD          = $(CROSS_COMPILE)ld
CC          = $(CROSS_COMPILE)gcc
CPP          = $(CC) -E
AR          = $(CROSS_COMPILE)ar
NM          = $(CROSS_COMPILE)nm
STRIP          = $(CROSS_COMPILE)strip
OBJCOPY          = $(CROSS_COMPILE)objcopy
OBJDUMP          = $(CROSS_COMPILE)objdump
AWK          = awk
...
...
...
```

在这些定义好的变量后面，我们又定义了两个变量： `USERINCLUDE` 和 `LINUXINCLUDE` 。他们包含了头文件的路径（第一个是给用户用的，第二个是给内核用的）：

```
USERINCLUDE    := \
        -I$(srctree)/arch/$(hdr-arch)/include/uapi \
        -Iarch/$(hdr-arch)/include/generated/uapi \
        -I$(srctree)/include/uapi \
        -Iinclude/generated/uapi \
        -include $(srctree)/include/linux/kconfig.h

LINUXINCLUDE    := \
        -I$(srctree)/arch/$(hdr-arch)/include \
        ...
```

以及给 C 编译器的标准标志：

```
KBUILD_CFLAGS   := -Wall -Wundef -Wstrict-prototypes -Wno-trigraphs \
           -fno-strict-aliasing -fno-common \
           -Werror-implicit-function-declaration \
           -Wno-format-security \
           -std=gnu89
```

这并不是最终确定的编译器标志，它们还可以在其他 Makefile 里面更新（比如 `arch/` 里面的 Kbuild ）。变量定义完之后，全部会被导出供其他 Makefile 使用。

下面的两个变量 `RCS_FIND_IGNORE` 和 `RCS_TAR_IGNORE` 包含了被版本控制系统忽略的文件：

```
export RCS_FIND_IGNORE := \( -name SCCS -o -name BitKeeper -o -name .svn -o    \
              -name CVS -o -name .pc -o -name .hg -o -name .git \) \
              -prune -o
export RCS_TAR_IGNORE := --exclude SCCS --exclude BitKeeper --exclude .svn \
             --exclude CVS --exclude .pc --exclude .hg --exclude .git
```

这就是全部了，我们已经完成了所有的准备工作，下一个点就是如何构建 `vmlinux` 。

直面内核构建

现在我们已经完成了所有的准备工作，根 Makefile（注：内核根目录下的 Makefile ）的下一步工作就是和编译内核相关的了。在这之前，我们不会在终端看到 `make` 命令输出的任何东西。但是现在编译的第一步开始了，这里我们需要从内核根 Makefile 的 598 行开始，这里可以看到目标 `vmlinux` ：

```
all: vmlinux
    include arch/$(SRCARCH)/Makefile
```

不要操心我们略过的从 `export RCS_FIND_IGNORE.....` 到 `all: vmlinux.....` 这一部分 Makefile 代码，他们只是负责根据各种配置文件（ `make *.config` ）生成不同目标内核的，因为之前我就说了这一部分我们只讨论构建内核的通用途径。

目标 `all:` 是在命令行如果不指定具体目标时默认使用的目标。你可以看到这里包含了架构相关的 Makefile（在这里就指的是 arch/x86/Makefile ）。从这一时刻起，我们会从这个 Makefile 继续进行下去。如我们所见，目标 `all` 依赖于根 Makefile 后面声明的 `vmlinux` ：

```
vmlinux: scripts/link-vmlinux.sh $(vmlinux-deps) FORCE
```

`vmlinux` 是 linux 内核的静态链接可执行文件格式。脚本 scripts/link-vmlinux.sh 把不同的编译好的子模块链接到一起形成了 vmlinux 。

第二个目标是 `vmlinux-deps` ，它的定义如下：

```
vmlinux-deps := $(KBUILD_LDS) $(KBUILD_VMLINUX_INIT) $(KBUILD_VMLINUX_MAIN)
```

它是由内核代码下的每个顶级目录的 `built-in.o` 组成的。之后我们还会检查内核所有的目录， `Kbuild` 会编译各个目录下所有的对应 `$(obj-y)` 的源文件。接着调用 `$(LD) -r` 把这些文件合并到一个 `build-in.o` 文件里。此时我们还没有 `vmlinux-deps` ，所以目标 `vmlinux` 现在还不会被构建。对我而言 `vmlinux-deps` 包含下面的文件：

```
arch/x86/kernel/vmlinux.lds arch/x86/kernel/head_64.o
arch/x86/kernel/head64.o    arch/x86/kernel/head.o
init/built-in.o             usr/built-in.o
arch/x86/built-in.o         kernel/built-in.o
mm/built-in.o               fs/built-in.o
ipc/built-in.o              security/built-in.o
crypto/built-in.o           block/built-in.o
lib/lib.a                   arch/x86/lib/lib.a
lib/built-in.o              arch/x86/lib/built-in.o
drivers/built-in.o          sound/built-in.o
firmware/built-in.o         arch/x86/pci/built-in.o
arch/x86/power/built-in.o   arch/x86/video/built-in.o
net/built-in.o
```

下一个可以被执行的目标如下：

```
$(sort $(vmlinux-deps)): $(vmlinux-dirs) ;
$(vmlinux-dirs): prepare scripts
    $(Q)$(MAKE) $(build)=$@
```

就像我们看到的，`vmlinux-dir` 依赖于两部分：`prepare` 和 `scripts` 。第一个 `prepare` 定义在内核的根 `Makefile` 中，准备工作分成三个阶段：

```
prepare: prepare0
prepare0: archprepare FORCE
    $(Q)$(MAKE) $(build)=.
archprepare: archheaders archscripts prepare1 scripts_basic

prepare1: prepare2 $(version_h) include/generated/utsrelease.h \
                   include/config/auto.conf
    $(cmd_crmodverdir)
prepare2: prepare3 outputmakefile asm-generic
```

第一个 `prepare0` 展开到 `archprepare` ，后者又展开到 `archheader` 和 `archscripts` ，这两个变量定义在 `x86_64` 相关的 Makefile 。让我们看看这个文件。`x86_64` 特定的 Makefile 从变量定义开始，这些变量都是和特定架构的配置文件 (defconfig，等等)有关联。在定义了编译 16-bit 代码的编译选项之后，根据变量 `BITS` 的值，如果是 `32` ，汇编代码、链接器、以及其它很多东西（全部的定义都可以在arch/x86/Makefile找到）对应的参数就是 `i386` ，而 `64` 就对应的是 `x86_84` 。

第一个目标是 Makefile 生成的系统调用列表(syscall table)中的 `archheaders` ：

```
archheaders:
    $(Q)$(MAKE) $(build)=arch/x86/entry/syscalls all
```

第二个目标是 Makefile 里的 `archscripts` ：

```
archscripts: scripts_basic
	$(Q)$(MAKE) $(build)=arch/x86/tools relocs
```

我们可以看到 `archscripts` 是依赖于根 Makefile 里的 `scripts_basic` 。首先我们可以看出 `scripts_basic` 是按照 scripts/basic 的 Makefile 执行 make 的：

```
scripts_basic:
	$(Q)$(MAKE) $(build)=scripts/basic
```

`scripts/basic/Makefile` 包含了编译两个主机程序 `fixdep` 和 `bin2` 的目标：

```
hostprogs-y	:= fixdep
hostprogs-$(CONFIG_BUILD_BIN2C)	+= bin2c
always		:= $(hostprogs-y)

$(addprefix $(obj)/,$(filter-out fixdep,$(always))): $(obj)/fixdep
```

第一个工具是 `fixdep` ：用来优化 gcc 生成的依赖列表，然后在重新编译源文件的时候告诉 `make` 。第二个工具是 `bin2c` ，它依赖于内核配置选项 `CONFIG_BUILD_BIN2C` ，并且它是一个用来将标准输入接口（LCTT 译注：即 stdin）收到的二进制流通过标准输出接口（即：stdout）转换成 C 头文件的非常小的 C 程序。你可能注意到这里有些奇怪的标志，如 `hostprogs-y` 等。这个标志用于所有的 `Kbuild` 文件，更多的信息你可以从 documentation 获得。在我们这里， `hostprogs-y` 告诉 `Kbuild` 这里有个名为 `fixed` 的程序，这个程序会通过和 `Makefile` 相同目录的 `fixdep.c` 编译而来。

执行 `make` 之后，终端的第一个输出就是 `Kbuild` 的结果：

```
$ make
  HOSTCC  scripts/basic/fixdep
```

当目标 `script_basic` 被执行，目标 `archscripts` 就会 `make` arch/x86/tools 下的 Makefile 和目标 `relocs` ：

```
$(Q)$(MAKE) $(build)=arch/x86/tools relocs
```

包含了重定位的信息的代码 `relocs_32.c` 和 `relocs_64.c` 将会被编译，这可以在 `make` 的输出中看到：

```
  HOSTCC  arch/x86/tools/relocs_32.o
  HOSTCC  arch/x86/tools/relocs_64.o
  HOSTCC  arch/x86/tools/relocs_common.o
  HOSTLD  arch/x86/tools/relocs
```

在编译完 `relocs.c` 之后会检查 `version.h` ：

```
$(version_h): $(srctree)/Makefile FORCE
    $(call filechk,version.h)
    $(Q)rm -f $(old_version_h)
```

我们可以在输出看到它：

```
CHK     include/config/kernel.release
```

以及在内核的根 Makefile 使用 `arch/x86/include/generated/asm` 的目标 `asm-generic` 来构建 `generic` 汇编头文件。在目标 `asm-generic` 之后， `archprepare` 就完成了，所以目标 `prepare0` 会接着被执行，如我上面所写：

```
prepare0: archprepare FORCE
    $(Q)$(MAKE) $(build)=.
```

注意 `build` ，它是定义在文件 scripts/Kbuild.include ，内容是这样的：

```
build := -f $(srctree)/scripts/Makefile.build obj
```

或者在我们的例子中，它就是当前源码目录路径- `.` ：

```
$(Q)$(MAKE) -f $(srctree)/scripts/Makefile.build obj=.
```

脚本 scripts/Makefile.build 通过参数 `obj` 给定的目录找到 `Kbuild` 文件，然后引入 `Kbuild` 文件：

```
include $(kbuild-file)
```

并根据这个构建目标。我们这里 `.` 包含了生成 `kernel/bounds.s` 和 `arch/x86/kernel/asm-offsets.s` 的 Kbuild 文件。在此之后，目标 `prepare` 就完成了它的工作。 `vmlinux-dirs` 也依赖于第二个目标 `scripts` ，它会编译接下来的几个程序： `filealias` ， `mk_elfconfig` ， `modpost` 等等。之后， `scripts/host-programs` 就可以开始编译我们的目标 `vmlinux-dirs` 了。

首先，我们先来理解一下 `vmlinux-dirs` 都包含了那些东西。在我们的例子中它包含了下列内核目录的路径：

```
init usr arch/x86 kernel mm fs ipc security crypto block
drivers sound firmware arch/x86/pci arch/x86/power
arch/x86/video net lib arch/x86/lib
```

我们可以在内核的根 Makefile 里找到 `vmlinux-dirs` 的定义：

```
vmlinux-dirs    := $(patsubst %/,%,$(filter %/, $(init-y) $(init-m) \
             $(core-y) $(core-m) $(drivers-y) $(drivers-m) \
             $(net-y) $(net-m) $(libs-y) $(libs-m)))

init-y        := init/
drivers-y    := drivers/ sound/ firmware/
net-y        := net/
libs-y        := lib/
...
...
...
```

这里我们借助函数 `patsubst` 和 `filter` 去掉了每个目录路径里的符号 `/` ，并且把结果放到 `vmlinux-dirs` 里。所以我们就有了 `vmlinux-dirs` 里的目录列表，以及下面的代码：

```
$(vmlinux-dirs): prepare scripts
    $(Q)$(MAKE) $(build)=$@
```

符号 `$@` 在这里代表了 `vmlinux-dirs` ，这就表明程序会递归遍历从 `vmlinux-dirs` 以及它内部的全部目录（依赖于配置），并且在对应的目录下执行 `make` 命令。我们可以在输出看到结果：

```
  CC      init/main.o
  CHK     include/generated/compile.h
  CC      init/version.o
  CC      init/do_mounts.o
  ...
  CC      arch/x86/crypto/glue_helper.o
  AS      arch/x86/crypto/aes-x86_64-asm_64.o
  CC      arch/x86/crypto/aes_glue.o
  ...
  AS      arch/x86/entry/entry_64.o
  AS      arch/x86/entry/thunk_64.o
  CC      arch/x86/entry/syscall_64.o
```

每个目录下的源代码将会被编译并且链接到 `built-io.o` 里：

```
$ find . -name built-in.o
./arch/x86/crypto/built-in.o
./arch/x86/crypto/sha-mb/built-in.o
./arch/x86/net/built-in.o
./init/built-in.o
./usr/built-in.o
...
...
```

好了，所有的 `built-in.o` 都构建完了，现在我们回到目标 `vmlinux` 上。你应该还记得，目标 `vmlinux` 是在内核的根 Makefile 里。在链接 `vmlinux` 之前，系统会构建 samples , Documentation 等等，但是如上文所述，我不会在本文描述这些。

```
vmlinux: scripts/link-vmlinux.sh $(vmlinux-deps) FORCE
    ...
    ...
    +$(call if_changed,link-vmlinux)
```

你可以看到，调用脚本 scripts/link-vmlinux.sh 的主要目的是把所有的 `built-in.o` 链接成一个静态可执行文件，和生成 System.map 。最后我们来看看下面的输出：

```
  LINK    vmlinux
  LD      vmlinux.o
  MODPOST vmlinux.o
  GEN     .version
  CHK     include/generated/compile.h
  UPD     include/generated/compile.h
  CC      init/version.o
  LD      init/built-in.o
  KSYM    .tmp_kallsyms1.o
  KSYM    .tmp_kallsyms2.o
  LD      vmlinux
  SORTEX  vmlinux
  SYSMAP  System.map
```

`vmlinux` 和 `System.map` 生成在内核源码树根目录下。

```
$ ls vmlinux System.map
System.map  vmlinux
```

这就是全部了， `vmlinux` 构建好了，下一步就是创建 bzImage.

制作bzImage

`bzImage` 就是压缩了的 linux 内核镜像。我们可以在构建了 `vmlinux` 之后通过执行 `make bzImage` 获得 `bzImage` 。同时我们可以仅仅执行 `make` 而不带任何参数也可以生成 `bzImage` ，因为它是在 arch/x86/kernel/Makefile 里预定义的、默认生成的镜像：

```
all: bzImage
```

让我们看看这个目标，它能帮助我们理解这个镜像是怎么构建的。我已经说过了 `bzImage` 是被定义在 arch/x86/kernel/Makefile ，定义如下：

```
bzImage: vmlinux
	$(Q)$(MAKE) $(build)=$(boot) $(KBUILD_IMAGE)
	$(Q)mkdir -p $(objtree)/arch/$(UTS_MACHINE)/boot
	$(Q)ln -fsn ../../x86/boot/bzImage $(objtree)/arch/$(UTS_MACHINE)/boot/$@
```

在这里我们可以看到第一次为 boot 目录执行 `make` ，在我们的例子里是这样的：

```
boot := arch/x86/boot
```

现在的主要目标是编译目录 `arch/x86/boot` 和 `arch/x86/boot/compressed` 的代码，构建 `setup.bin` 和 `vmlinux.bin` ，最后用这两个文件生成 `bzImage` 。第一个目标是定义在 arch/x86/boot/Makefile 的 `$(obj)/setup.elf` ：

```
$(obj)/setup.elf: $(src)/setup.ld $(SETUP_OBJS) FORCE
	$(call if_changed,ld)
```

我们已经在目录 `arch/x86/boot` 有了链接脚本 `setup.ld` ，和扩展到 `boot` 目录下全部源代码的变量 `SETUP_OBJS` 。我们可以看看第一个输出：

```
  AS      arch/x86/boot/bioscall.o
  CC      arch/x86/boot/cmdline.o
  AS      arch/x86/boot/copy.o
  HOSTCC  arch/x86/boot/mkcpustr
  CPUSTR  arch/x86/boot/cpustr.h
  CC      arch/x86/boot/cpu.o
  CC      arch/x86/boot/cpuflags.o
  CC      arch/x86/boot/cpucheck.o
  CC      arch/x86/boot/early_serial_console.o
  CC      arch/x86/boot/edd.o
```

下一个源码文件是 arch/x86/boot/header.S ，但是我们不能现在就编译它，因为这个目标依赖于下面两个头文件：

```
$(obj)/header.o: $(obj)/voffset.h $(obj)/zoffset.h
```

第一个头文件 `voffset.h` 是使用 `sed` 脚本生成的，包含用 `nm` 工具从 `vmlinux` 获取的两个地址：

```
#define VO__end 0xffffffff82ab0000
#define VO__text 0xffffffff81000000
```

这两个地址是内核的起始和结束地址。第二个头文件 `zoffset.h` 在 arch/x86/boot/compressed/Makefile 可以看出是依赖于目标 `vmlinux` 的：

```
$(obj)/zoffset.h: $(obj)/compressed/vmlinux FORCE
    $(call if_changed,zoffset)
```

目标 `$(obj)/compressed/vmlinux` 依赖于 `vmlinux-objs-y` —— 说明需要编译目录 arch/x86/boot/compressed 下的源代码，然后生成 `vmlinux.bin` 、 `vmlinux.bin.bz2` ，和编译工具 `mkpiggy` 。我们可以在下面的输出看出来：

```
  LDS     arch/x86/boot/compressed/vmlinux.lds
  AS      arch/x86/boot/compressed/head_64.o
  CC      arch/x86/boot/compressed/misc.o
  CC      arch/x86/boot/compressed/string.o
  CC      arch/x86/boot/compressed/cmdline.o
  OBJCOPY arch/x86/boot/compressed/vmlinux.bin
  BZIP2   arch/x86/boot/compressed/vmlinux.bin.bz2
  HOSTCC  arch/x86/boot/compressed/mkpiggy
```

`vmlinux.bin` 是去掉了调试信息和注释的 `vmlinux` 二进制文件，加上了占用了 `u32` （LCTT 译注：即4-Byte）的长度信息的 `vmlinux.bin.all` 压缩后就是 `vmlinux.bin.bz2` 。其中 `vmlinux.bin.all` 包含了 `vmlinux.bin` 和 `vmlinux.relocs` （LCTT 译注：vmlinux 的重定位信息），其中 `vmlinux.relocs` 是 `vmlinux` 经过程序 `relocs` 处理之后的 `vmlinux` 镜像（见上文所述）。我们现在已经获取到了这些文件，汇编文件 `piggy.S` 将会被 `mkpiggy` 生成、然后编译：

```
  MKPIGGY arch/x86/boot/compressed/piggy.S
  AS      arch/x86/boot/compressed/piggy.o
```

这个汇编文件会包含经过计算得来的、压缩内核的偏移信息。处理完这个汇编文件，我们就可以看到 `zoffset` 生成了：

```
  ZOFFSET arch/x86/boot/zoffset.h
```

现在 `zoffset.h` 和 `voffset.h` 已经生成了，arch/x86/boot 里的源文件可以继续编译：

```
AS      arch/x86/boot/header.o
CC      arch/x86/boot/main.o
CC      arch/x86/boot/mca.o
CC      arch/x86/boot/memory.o
CC      arch/x86/boot/pm.o
AS      arch/x86/boot/pmjump.o
CC      arch/x86/boot/printf.o
CC      arch/x86/boot/regs.o
CC      arch/x86/boot/string.o
CC      arch/x86/boot/tty.o
CC      arch/x86/boot/video.o
CC      arch/x86/boot/video-mode.o
CC      arch/x86/boot/video-vga.o
CC      arch/x86/boot/video-vesa.o
CC      arch/x86/boot/video-bios.o
```

所有的源代码会被编译，他们最终会被链接到 `setup.elf` ：

```
LD      arch/x86/boot/setup.elf
```

或者：

```
ld -m elf_x86_64   -T arch/x86/boot/setup.ld arch/x86/boot/a20.o arch/x86/boot/bioscall.o arch/x86/boot/cmdline.o arch/x86/boot/copy.o arch/x86/boot/cpu.o arch/x86/boot/cpuflags.o arch/x86/boot/cpucheck.o arch/x86/boot/early_serial_console.o arch/x86/boot/edd.o arch/x86/boot/header.o arch/x86/boot/main.o arch/x86/boot/mca.o arch/x86/boot/memory.o arch/x86/boot/pm.o arch/x86/boot/pmjump.o arch/x86/boot/printf.o arch/x86/boot/regs.o arch/x86/boot/string.o arch/x86/boot/tty.o arch/x86/boot/video.o arch/x86/boot/video-mode.o arch/x86/boot/version.o arch/x86/boot/video-vga.o arch/x86/boot/video-vesa.o arch/x86/boot/video-bios.o -o arch/x86/boot/setup.elf
```

最后的两件事是创建包含目录 `arch/x86/boot/*` 下的编译过的代码的 `setup.bin` ：

```
objcopy  -O binary arch/x86/boot/setup.elf arch/x86/boot/setup.bin
```

以及从 `vmlinux` 生成 `vmlinux.bin` ：

```
objcopy  -O binary -R .note -R .comment -S arch/x86/boot/compressed/vmlinux arch/x86/boot/vmlinux.bin
```

最最后，我们编译主机程序 arch/x86/boot/tools/build.c ，它将会用来把 `setup.bin` 和 `vmlinux.bin` 打包成 `bzImage` ：

```
arch/x86/boot/tools/build arch/x86/boot/setup.bin arch/x86/boot/vmlinux.bin arch/x86/b
oot/zoffset.h arch/x86/boot/bzImage
```

实际上 `bzImage` 就是把 `setup.bin` 和 `vmlinux.bin` 连接到一起。最终我们会看到输出结果，就和那些用源码编译过内核的同行的结果一样：

```
Setup is 16268 bytes (padded to 16384 bytes).
System is 4704 kB
CRC 94a88f9a
Kernel: arch/x86/boot/bzImage is ready  (#5)
```

全部结束。

结论

这就是本文的结尾部分。本文我们了解了编译内核的全部步骤：从执行 `make` 命令开始，到最后生成 `bzImage` 。我知道，linux 内核的 Makefile 和构建 linux 的过程第一眼看起来可能比较迷惑，但是这并不是很难。希望本文可以帮助你理解构建 linux 内核的整个流程。

注： 本文由 LCTT 原创翻译，Linux中国 荣誉推出

链接

- GNU make util
- Linux kernel top Makefile
- cross-compilation
- Ctags
- sparse
- bzImage
- uname
- shell
- Kbuild
- binutils
- gcc
- Documentation
- System.map
- Relocation

介绍

在写 linux-insides 一书的过程中，我收到了很多邮件询问关于链接器和链接器脚本的问题。所以我决定写这篇文章来介绍链接器和目标文件的链接方面的知识。

如果我们打开维基百科的 `链接器` 页，我们将会看到如下定义：

> 在计算机科学中，链接器（英文：Linker），是一个计算机程序，它将一个或多个由编译器生成的目标文件链接为一个单独的可执行文件，库文件或者另外一个目标文件

如果你曾经用 C 写过至少一个程序，那你就已经见过以 `*.o` 扩展名结尾的文件了。这些文件是目标文件。目标文件是一块块的机器码和数据，其数据包含了引用其他目标文件或库的数据和函数的占位符地址，也包括其自身的函数和数据列表。链接器的主要目的就是收集/处理每一个目标文件的代码和数据，将它们转成最终的可执行文件或者库。在这篇文章里，我们会试着研究这个流程的各个方面。开始吧。

链接流程

让我们按以下结构创建一个项目：

```
*-linkers
*--main.c
*--lib.c
*--lib.h
```

我们的 `main.c` 源文件包含了：

```
#include <stdio.h>

#include "lib.h"

int main(int argc, char **argv) {
    printf("factorial of 5 is: %d\n", factorial(5));
    return 0;
}
```

`lib.c` 文件包含了：

```
int factorial(int base) {
    int res,i = 1;

    if (base == 0) {
        return 1;
    }

    while (i <= base) {
        res *= i;
        i++;
    }

    return res;
}
```

`lib.h` 文件包含了：

```
#ifndef LIB_H
#define LIB_H

int factorial(int base);

#endif
```

现在让我们用以下命令单独编译 `main.c` 源码：

```
$ gcc -c main.c
```

如果我们用 `nm` 工具查看输出的目标文件，我们将会看到如下输出：

```
$ nm -A main.o
main.o:                  U factorial
main.o:0000000000000000 T main
main.o:                  U printf
```

`nm` 工具让我们能够看到给定目标文件的符号表列表。其包含了三列：第一列是该目标文件的名称和解析得到的符号地址。第二列包含了一个表示该符号状态的字符。这里 `U` 表示 `未定义`，`T` 表示该符号被置于 `.text` 段。在这里，`nm` 工具向我们展示了 `main.c` 文件里包含的三个符号：

- `factorial` - 在 `lib.c` 文件中定义的阶乘函数。因为我们只编译了 `main.c`，所以其不知道任何有关 `lib.c` 文件的事；
- `main` - 主函数;
- `printf` - 来自 glibc 库的函数。`main.c` 同样不知道任何与其相关的事。

目前我们可以从 `nm` 的输出中了解哪些事情呢？ `main.o` 目标文件包含了在地址 `0000000000000000` 处的本地变量 `main` （在被链接后其将会被赋予正确的地址），以及两个无法解析的符号。我们可以从 `main.o` 的反汇编输出中了解这些信息：

```
$ objdump -S main.o

main.o:     file format elf64-x86-64
Disassembly of section .text:

0000000000000000 <main>:
   0:    55                      push   %rbp
   1:    48 89 e5                mov    %rsp,%rbp
   4:    48 83 ec 10             sub    $0x10,%rsp
   8:    89 7d fc                mov    %edi,-0x4(%rbp)
   b:    48 89 75 f0             mov    %rsi,-0x10(%rbp)
   f:    bf 05 00 00 00          mov    $0x5,%edi
  14:    e8 00 00 00 00          callq  19 <main+0x19>
  19:    89 c6                   mov    %eax,%esi
  1b:    bf 00 00 00 00          mov    $0x0,%edi
  20:    b8 00 00 00 00          mov    $0x0,%eax
  25:    e8 00 00 00 00          callq  2a <main+0x2a>
  2a:    b8 00 00 00 00          mov    $0x0,%eax
  2f:    c9                      leaveq
  30:    c3                      retq
```

这里我们只关注两个 `callq` 操作。这两个 `callq` 操作包含了 `链接器存根`，或者函数的名称和其相对当前的下一条指令的偏移。这些存根将会被更新到函数的真实地址。我们可以在下面的 `objdump` 输出看到这些函数的名字：

```
$ objdump -S -r main.o

...
  14:    e8 00 00 00 00          callq  19 <main+0x19>
  15: R_X86_64_PC32                  factorial-0x4
  19:    89 c6                   mov    %eax,%esi
...
  25:    e8 00 00 00 00          callq  2a <main+0x2a>
  26:   R_X86_64_PC32                    printf-0x4
  2a:    b8 00 00 00 00          mov    $0x0,%eax
...
```

`objdump` 工具中的 `-r` 或 `--reloc` 选项会打印文件的 `重定位` 条目。现在让我们更加深入重定位流程。

重定位

重定位是连接符号引用和符号定义的流程。让我们看看前一段 `objdump` 的输出：

```
14:    e8 00 00 00 00          callq  19 <main+0x19>
15:   R_X86_64_PC32                  factorial-0x4
19:    89 c6                   mov    %eax,%esi
```

注意第一行的 `e8 00 00 00 00` 。`e8` 是 `call` 的 操作码 ，这一行的剩余部分是一个相对偏移。所以 `e8 00 00 00` 包含了一个单字节操作码，跟着一个四字节地址。注意 `00 00 00 00` 是 4 个字节。为什么只有 4 字节 而不是 `x86_64` 64 位机器上的 8 字节地址？其实我们用了 `-mcmodel=small` 选项来编译 `main.c` ！从 `gcc` 的指南上看：

```
-mcmodel=small

为小代码模型生成代码: 目标程序及其符号必须被链接到低于 2GB 的地址空间。指针是 64 位的。程序可以被动态或静态的链接。这是默认的代码模型。
```

当然，我们在编译时并没有将这一选项传给 `gcc` ，但是这是默认的。从上面摘录的 `gcc` 指南我们知道，我们的程序会被链接到低于 2 GB 的地址空间。因此 4 字节已经足够。所以我们有了 `call` 指令和一个未知的地址。当我们编译 `main.c` 以及它的依赖形成一个可执行文件时，关注阶乘函数的调用，我们看到：

```
$ gcc main.c lib.c -o factorial | objdump -S factorial | grep factorial

factorial:     file format elf64-x86-64
...
...
0000000000400506 <main>:
    40051a:    e8 18 00 00 00          callq  400537 <factorial>
...
...
0000000000400537 <factorial>:
    400550:    75 07                   jne    400559 <factorial+0x22>
    400557:    eb 1b                   jmp    400574 <factorial+0x3d>
    400559:    eb 0e                   jmp    400569 <factorial+0x32>
    40056f:    7e ea                   jle    40055b <factorial+0x24>
...
...
```

在前面的输出中我们可以看到， `main` 函数的地址是 `0x0000000000400506` 。为什么它不是从 `0x0` 开始的呢？你可能已经知道标准 C 程序是使用 `glibc` 的 C 标准库链接的（假设参数 `-nostdlib` 没有被传给 `gcc` ）。编译后的程序代码包含了用于在程序启动时初始化程序中数据的构造函数。这些函数需要在程序启动前被调用，或者说在 `main` 函数之前被调用。为了让初始化和终止函数起作用，编译器必须在汇编代码中输出一些让这些函数在正确时间被调用的代码。执行这个程序将会启动位于特殊的 `.init` 段的代码。我们可以从以下的 objdump 输出中看出：

```
objdump -S factorial | less

factorial:     file format elf64-x86-64

Disassembly of section .init:

00000000004003a8 <_init>:
  4003a8:       48 83 ec 08             sub    $0x8,%rsp
  4003ac:       48 8b 05 a5 05 20 00    mov    0x2005a5(%rip),%rax        # 600958 <_D
YNAMIC+0x1d0>
```

注意其开始于相对 `glibc` 代码偏移 `0x00000000004003a8` 的地址。我们也可以运行 `readelf` ，在 ELF 输出中检查：

```
$ readelf -d factorial | grep \(INIT\)
 0x000000000000000c (INIT)               0x4003a8
```

所以， `main` 函数的地址是 `0000000000400506` ，为相对于 `.init` 段的偏移地址。我们可以从输出中看出， `factorial` 函数的地址是 `0x0000000000400537` ，并且现在调用 `factorial` 函数的二进制代码是 `e8 18 00 00 00` 。我们已经知道 `e8` 是 `call` 指令的操作码，接下来的 `18 00 00 00` （注意 `x86_64` 中地址是小头存储的，所以是 `00 00 00 18` ）是从 `callq` 到 `factorial` 函数的偏移。

```
>>> hex(0x40051a + 0x18 + 0x5) == hex(0x400537)
True
```

所以我们把 `0x18` 和 `0x5` 加到 `call` 指令的地址上。偏移是从接下来一条指令开始算起的。我们的调用指令是 5 字节长（ `e8 18 00 00 00` ）并且 `0x18` 是从 `factorial` 函数之后的调用算起的偏移。编译器一般按程序地址从零开始创建目标文件。但是如果程序由多个目标文件生成，这些地址会重叠。

我们在这一段看到的是 `重定位` 流程。这个流程为程序中各个部分赋予加载地址，调整程序中的代码和数据以反映出赋值的地址。

好了，现在我们知道了一点关于链接器和重定位的知识，是时候通过链接我们的目标文件来来学习更多关于链接器的知识了。

GNU 链接器

如标题所说，在这篇文章中，我将会使用 GNU 链接器 或者说 `ld` 。当然我们可以使用 `gcc` 来链接我们的 `factorial` 项目：

```
$ gcc main.c lib.o -o factorial
```

在这之后，作为结果我们将会得到可执行文件—— `factorial` ：

```
./factorial
factorial of 5 is: 120
```

但是 `gcc` 不会链接目标文件。取而代之，其会使用 `GUN ld` 链接器的包装—— `collect2` 。

```
~$ /usr/lib/gcc/x86_64-linux-gnu/4.9/collect2 --version
collect2 version 4.9.3
/usr/bin/ld --version
GNU ld (GNU Binutils for Debian) 2.25
...
...
...
```

好，我们可以使用 gcc 并且其会为我们的程序生成可执行文件。但是让我们看看如何使用 `GUN ld` 实现相同的目的。首先，让我们尝试用如下样例链接这些目标文件：

```
ld main.o lib.o -o factorial
```

尝试一下，你将会得到如下错误：

```
$ ld main.o lib.o -o factorial
ld: warning: cannot find entry symbol _start; defaulting to 00000000004000b0
main.o: In function `main':
main.c:(.text+0x26): undefined reference to `printf'
```

这里我们可以看到两个问题：

- 链接器无法找到 `_start` 符号；
- 链接器对 `printf` 一无所知。

首先，让我们尝试理解好像是我们程序运行所需要的 `_start` 入口符号是什么。当我开始学习编程时，我知道了 `main` 函数是程序的入口点。我认为你们也是如此认为的 :) 但实际上这不是入口点， `_start` 才是。 `_start` 符号被 `crt1.o` 所定义。我们可以用如下指令发现它：

```
$ objdump -S /usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crt1.o

/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crt1.o:     file format el
f64-x86-64


Disassembly of section .text:

0000000000000000 <_start>:
   0:	31 ed                	xor    %ebp,%ebp
   2:	49 89 d1             	mov    %rdx,%r9
   ...
   ...
   ...
```

我们将该目标文件作为第一个参数传递给 `ld` 指令（如上所示）。现在让我们尝试链接它，会得到如下结果：

```
ld /usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crt1.o \
main.o lib.o -o factorial

/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crt1.o: In function `_star
t':
/tmp/buildd/glibc-2.19/csu/../sysdeps/x86_64/start.S:115: undefined reference to `__li
bc_csu_fini'
/tmp/buildd/glibc-2.19/csu/../sysdeps/x86_64/start.S:116: undefined reference to `__li
bc_csu_init'
/tmp/buildd/glibc-2.19/csu/../sysdeps/x86_64/start.S:122: undefined reference to `__li
bc_start_main'
main.o: In function `main':
main.c:(.text+0x26): undefined reference to `printf'
```

不幸的是，我们甚至会看到更多报错。我们可以在这里看到关于未定义 `printf` 的旧错误以及另外三个未定义的引用：

- `__libc_csu_fini`
- `__libc_csu_init`
- `__libc_start_main`

`_start` 符号被定义在 `glibc` 源文件的汇编文件 sysdeps/x86_64/start.S 中。我们可以在那里找到如下汇编代码：

```
mov $__libc_csu_fini, %R8_LP
mov $__libc_csu_init, %RCX_LP
...
call __libc_start_main
```

这里我们传递了 `.init` 和 `.fini` 段的入口点地址，它们包含了程序开始和结束时被执行的代码。并且在结尾我们看到对我们程序的 `main` 函数的调用。这三个符号被定义在源文件 csu/elf-init.c 中。如下两个目标文件：

- `crtn.o`；
- `crti.o`.

定义了 .init 和 .fini 段的开端和尾声（分别为符号 `_init` 和 `_fini` ）。

`crtn.o` 目标文件包含了 `.init` 和 `.fini` 这些段：

```
$ objdump -S /usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crtn.o

0000000000000000 <.init>:
   0:	48 83 c4 08          	add    $0x8,%rsp
   4:	c3                   	retq

Disassembly of section .fini:

0000000000000000 <.fini>:
   0:	48 83 c4 08          	add    $0x8,%rsp
   4:	c3                   	retq
```

且 `crti.o` 目标文件包含了符号 `_init` 和 `_fini` 。让我们再次尝试链接这两个目标文件：

```
$ ld \
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crt1.o \
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crti.o \
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crtn.o main.o lib.o \
-o factorial
```

当然，我们会得到相同的错误。现在我们需要把 `-lc` 选项传递给 `ld` 。这个选项将会在环境变量 `$LD_LIBRARY_PATH` 指定的目录中搜索标准库。让我们再次尝试用 `-lc` 选项链接：

```
$ ld \
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crt1.o \
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crti.o \
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crtn.o main.o lib.o -lc \
-o factorial
```

最后我们获得了一个可执行文件，但是如果我们尝试运行它，我们会遇到奇怪的结果：

```
$ ./factorial
bash: ./factorial: No such file or directory
```

这里除了什么问题？让我们用 readelf 工具看看这个可执行文件：

```
$ readelf -l factorial

Elf file type is EXEC (Executable file)
Entry point 0x4003c0
There are 7 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x0000000000000188 0x0000000000000188  R E    8
  INTERP         0x00000000000001c8 0x00000000004001c8 0x00000000004001c8
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000610 0x0000000000000610  R E    200000
  LOAD           0x0000000000000610 0x0000000000600610 0x0000000000600610
                 0x00000000000001cc 0x00000000000001cc  RW     200000
  DYNAMIC        0x0000000000000610 0x0000000000600610 0x0000000000600610
                 0x0000000000000190 0x0000000000000190  RW     8
  NOTE           0x00000000000001e4 0x00000000004001e4 0x00000000004001e4
                 0x0000000000000020 0x0000000000000020  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_r .rel
a.dyn .rela.plt .init .plt .text .fini .rodata .eh_frame
   03     .dynamic .got .got.plt .data
   04     .dynamic
   05     .note.ABI-tag
   06
```

注意这奇怪的一行：

```
  INTERP         0x00000000000001c8 0x00000000004001c8 0x00000000004001c8
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
```

`elf` 文件的 `.interp` 段保存了一个程序解释器的路径名，或者说 `.interp` 段就包含了一个动态链接器名字的 `ascii` 字符串。动态链接器是 Linux 的一部分，其通过将库的内容从磁盘复制到内存中以加载和链接一个可执行文件被执行所需要的动态链接库。我们可以从 `readelf` 命令的输出中看到，针对 `x86_64` 架构，其被放在 `/lib64/ld-linux-x86-64.so.2` 。现在让我们把 `ld-linux-x86-64.so.2` 的路径和 `-dynamic-linker` 选项一起传递给 `ld` 调用，然后会看到如下结果：

```
$ gcc -c main.c lib.c

$ ld \
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crt1.o \
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crti.o \
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crtn.o main.o lib.o \
-dynamic-linker /lib64/ld-linux-x86-64.so.2 \
-lc -o factorial
```

现在我们可以像普通可执行文件一样执行它了：

```
$ ./factorial

factorial of 5 is: 120
```

成功了！在第一行，我们把源文件 `main.c` 和 `lib.c` 编译成目标文件。执行 `gcc` 之后我们将会获得 `main.o` 和 `lib.o` ：

```
$ file lib.o main.o
lib.o:  ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
main.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

在这之后，我们用所需的系统目标文件和库连链接我们的程序。我们刚看了一个简单的关于如何用 `gcc` 编译器和 `GNU ld` 链接器编译和链接一个 C 程序的样例。在这个样例中，我们使用了一些 `GNU linker` 的命令行选项，但是除了 `-o` 、 `-dynamic-linker` 等，它还支持其他很多选项。此外， `GNU ld` 还拥有其自己的语言来控制链接过程。在接下来的两个段落中我们深入讨论。

实用的 GNU 链接器命令行选项

正如我之前所说，你也可以从 `GNU linker` 的指南看到，其拥有大量的命令行选项。我们已经在这篇文章见到一些： `-o <output>` - 告诉 `ld` 将链接结果输出成一个叫做 `output` 的文件， `-l<name>` - 通过文件名添加指定存档或者目标文件， `-dynamic-linker` 通过名字指定动态链接器。当然， `ld` 支持更多选项，让我们看看其中的一些。

第一个实用的选项是 `@file` 。在这里 `file` 指定了命令行选项将读取的文件名。比如我们可以创建一个叫做 `linker.ld` 的文件，把我们上一个例子里面的命令行参数放进去然后执行：

```
$ ld @linker.ld
```

下一个命令行选项是 `-b` 或 `--format`。这个命令行选项指定了输入的目标文件的格式是 `ELF`, `DJGPP/COFF` 等。针对输出文件也有相同功能的选项 `--oformat=output-format`。

下一个命令行选项是 `--defsym`。该选项的完整格式是 `--defsym=symbol=expression`。它允许在输出文件中创建包含了由表达式给出了绝对地址的全局符号。在下面的例子中，我们会发现这个命令行选项很实用：在 Linux 内核源码中关于 ARM 架构内核解压的 Makefile - arch/arm/boot/compressed/Makefile，我们可以找到如下定义：

```
LDFLAGS_vmlinux = --defsym _kernel_bss_size=$(KBSS_SZ)
```

正如我们所知，其在输出文件中用 `.bss` 段的大小定义了 `_kernel_bss_size` 符号。这个符号将会作为第一个 汇编文件 在内核解压阶段被执行：

```
ldr r5, =_kernel_bss_size
```

下一个选项是 `-shared`，其允许我们创建共享库。`-M` 或者说 `-map <filename>` 命令行选项会打印带符号信息的链接映射内容。在这里是：

```
$ ld -M @linker.ld
...
...
...
.text           0x00000000004003c0      0x112
 *(.text.unlikely .text.*_unlikely .text.unlikely.*)
 *(.text.exit .text.exit.*)
 *(.text.startup .text.startup.*)
 *(.text.hot .text.hot.*)
 *(.text .stub .text.* .gnu.linkonce.t.*)
 .text          0x00000000004003c0       0x2a /usr/lib/gcc/x86_64-linux-gnu/4.9/../../
../x86_64-linux-gnu/crt1.o
...
...
...
 .text          0x00000000004003ea       0x31 main.o
                0x00000000004003ea                main
 .text          0x000000000040041b       0x3f lib.o
                0x000000000040041b                factorial
```

当然，`GNU 链接器` 支持标准的命令行选项：`--help` 和 `--version` 能够打印 `ld` 的命令帮助、使用方法和版本。以上就是所有关于 `GNU 链接器` 命令行选项的内容。当然这不是 `ld` 工具支持的所有命令行选项。你可以在指南中找到 `ld` 工具的完整文档。

链接器控制语言

如我之前所说， `ld` 支持它自己的语言。它接受由一种 AT&T 链接器控制语法的超集编写的链接器控制语言文件，以提供对链接过程明确且完全的控制。接下来让我们关注其中细节。

我们可以通过链接器语言控制：

- 输入文件；
- 输出文件；
- 文件格式；
- 段的地址；
- 其他更多...

用链接器控制语言编写的命令通常被放在一个被称作链接器脚本的文件中。我们可以通过命令行选项 `-T` 将其传递给 `ld` 。一个链接器脚本的主要命令是 `SECTIONS` 指令。每个链接器脚本必须包含这个指令，并且其决定了输出文件的 `映射` 。特殊变量 `.` 包含了当前输出的位置。让我们写一个简单的汇编程序，然后看看如何使用链接器脚本来控制程序的链接。我们将会使用一个 hello world 程序作为样例。

```
.data
    msg    .ascii "hello, world!",`\n`

.text

    global    _start

_start:
    mov    $1,%rax
    mov    $1,%rdi
    mov    $msg,%rsi
    mov    $14,%rdx
    syscall

    mov    $60,%rax
    mov    $0,%rdi
    syscall
```

我们可以用以下命令编译并链接：

```
$ as -o hello.o hello.asm
$ ld -o hello hello.o
```

我们的程序包含了两个段： `.text` 包含了程序的代码， `.data` 段包含了被初始化的变量。让我们写一个简单的链接脚本，然后尝试用它来链接我们的 `hello.asm` 汇编文件。我们的脚本是：

```
/*
 * Linker script for the factorial
 */
OUTPUT(hello)
OUTPUT_FORMAT("elf64-x86-64")
INPUT(hello.o)

SECTIONS
{
    . = 0x200000;
    .text : {
          *(.text)
    }

    . = 0x400000;
    .data : {
          *(.data)
    }
}
```

在前三行你可以看到 `C` 风格的注释。之后是 `OUTPUT` 和 `OUTPUT_FORMAT` 命令，指定了我们的可执行文件名称和格式。下一个指令， `INPUT` ，指定了给 `ld` 的输入文件。接下来，我们可以看到主要的 `SECTIONS` 指令，正如我写的，它是必须存在于每个链接器脚本中。 `SECTIONS` 命令表示了输出文件中的段的集合和顺序。在 `SECTIONS` 命令的开头，我们可以看到一行 `. = 0x200000` 。我上面已经写过， `.` 命令指向输出中的当前位置。这一行说明代码段应该被加载到地址 `0x200000` 。 `. = 0x400000` 一行说明数据段应该被加载到地址 `0x400000` 。 `. = 0x200000` 之后的第二行定义 `.text` 作为输出段。我们可以看到其中的 `*(.text)` 表达式。 `*` 符号是一个匹配任意文件名的通配符。换句话说， `*(.text)` 表达式代表所有输入文件中的所有 `.text` 输入段。在我们的样例中，我们可以将其重写为 `hello.o(.text)` 。在地址计数器 `. = 0x400000` 之后，我们可以看到数据段的定义。

我们可以用以下语句进行编译和链接：

```
$ as -o hello.o hello.S && ld -T linker.script && ./hello
hello, world!
```

如果我们用 `objdump` 工具深入查看，我们可以看到 `.text` 段从地址 `0x200000` 开始， `.data` 段从 `0x400000` 开始：

```
$ objdump -D hello

Disassembly of section .text:

0000000000200000 <_start>:
  200000:    48 c7 c0 01 00 00 00     mov    $0x1,%rax
  ...

Disassembly of section .data:

0000000000400000 <msg>:
  400000:    68 65 6c 6c 6f           pushq  $0x6f6c6c65
  ...
```

除了我们已经看到的命令，另外还有一些。首先是 `ASSERT(exp, message)` ，保证给定的表达式不为零。如果为零，那么链接器会退出同时返回错误码，打印错误信息。如果你已经阅读了 linux-insides 的 Linux 内核启动流程，你或许知道 Linux 内核的设置头的偏移为 `0x1f1` 。在 Linux 内核的链接器脚本中，我们可以看到下面的校验：

```
. = ASSERT(hdr == 0x1f1, "The setup header has the wrong offset!");
```

`INCLUDE filename` 允许我们在当前的链接器脚本中包含外部符号。我们可以在一个链接器脚本中给一个符号赋值。 `ld` 支持一些赋值操作符：

- symbol = expression ;
- symbol += expression ;
- symbol -= expression ;
- symbol *= expression ;
- symbol /= expression ;
- symbol <<= expression ;
- symbol >>= expression ;
- symbol &= expression ;
- symbol |= expression ;

正如你注意到的，所有操作符都是 C 赋值操作符。比如我们可以在我们的链接器脚本中使用：

```
START_ADDRESS = 0x200000;
DATA_OFFSET   = 0x200000;

SECTIONS
{
    . = START_ADDRESS;
    .text : {
          *(.text)
    }

    . = START_ADDRESS + DATA_OFFSET;
    .data : {
          *(.data)
    }
}
```

你可能已经注意到了链接器脚本中表达式的语法和 C 表达式相同。除此之外，这个链接控制语言还支持如下内嵌函数：

- `ABSOLUTE` - 返回给定表达式的绝对值；
- `ADDR` - 接受段，返回其地址；
- `ALIGN` - 返回和给定表达式下一句的边界对齐的位置计数器（ `.` 操作符）的值；
- `DEFINED` - 如果给定符号在全局符号表中，返回 `1` ，否则 `0` ；
- `MAX` and `MIN` - 返回两个给定表达式中的最大、最小值；
- `NEXT` - 返回一个是当前表达式倍数的未分配地址；
- `SIZEOF` - 返回给定名字的段以字节计数的大小。

以上就是全部了。

总结

这是关于链接器文章的结尾。在这篇文章中，我们已经学习了很多关于链接器的知识，比如什么是链接器、为什么需要它、如何使用它等等...

如果你发现文中描述有任何问题，请提交一个 PR 到 linux-insides-zh 。

相关链接

- Book about Linux kernel insides
- linker
- object files
- glibc
- opcode

- ELF
- GNU linker
- My posts about assembly programming for x86_64
- readelf

用户空间的程序启动过程

简介

虽然 linux-insides-zh 大多描述的是内核相关的东西，但是我已经决定写一个大多与用户空间相关的部分。

系统调用章节的第四部分已经描述了当我们想运行一个程序，Linux 内核的行为。这部分我想研究一下从用户空间的角度，当我们在 Linux 系统上运行一个程序，会发生什么。

我不知道你知识储备如何，但是在我的大学时期我学到，一个 `C` 程序从一个叫做 main 的函数开始执行。而且，这是部分正确的。每时每刻，当我们开始写一个新的程序时，我们从下面的实例代码开始编程：

```
int main(int argc, char *argv[]) {
    // Entry point is here
}
```

但是你如何对于底层编程感兴趣的话，可能你已经知道 `main` 函数并不是程序的真正入口。如果你在调试器中看了下面这个简单程序，就可以很确信这一点：

```
int main(int argc, char *argv[]) {
    return 0;
}
```

让我们来编译并且在 gdb 中运行这个程序：

```
$ gcc -ggdb program.c -o program
$ gdb ./program
The target architecture is assumed to be i386:x86-64:intel
Reading symbols from ./program...done.
```

让我们在 gdb 中执行 `info files` 这个指令。这个指令会打印关于被不同段占据的内存和调试目标的信息。

```
(gdb) info files
Symbols from "/home/alex/program".
Local exec file:
    `/home/alex/program', file type elf64-x86-64.
    Entry point: 0x400430
    0x0000000000400238 - 0x0000000000400254 is .interp
    0x0000000000400254 - 0x0000000000400274 is .note.ABI-tag
    0x0000000000400274 - 0x0000000000400298 is .note.gnu.build-id
    0x0000000000400298 - 0x00000000004002b4 is .gnu.hash
    0x00000000004002b8 - 0x0000000000400318 is .dynsym
    0x0000000000400318 - 0x0000000000400357 is .dynstr
    0x0000000000400358 - 0x0000000000400360 is .gnu.version
    0x0000000000400360 - 0x0000000000400380 is .gnu.version_r
    0x0000000000400380 - 0x0000000000400398 is .rela.dyn
    0x0000000000400398 - 0x00000000004003c8 is .rela.plt
    0x00000000004003c8 - 0x00000000004003e2 is .init
    0x00000000004003f0 - 0x0000000000400420 is .plt
    0x0000000000400420 - 0x0000000000400428 is .plt.got
    0x0000000000400430 - 0x00000000004005e2 is .text
    0x00000000004005e4 - 0x00000000004005ed is .fini
    0x00000000004005f0 - 0x0000000000400610 is .rodata
    0x0000000000400610 - 0x0000000000400644 is .eh_frame_hdr
    0x0000000000400648 - 0x000000000040073c is .eh_frame
    0x0000000000600e10 - 0x0000000000600e18 is .init_array
    0x0000000000600e18 - 0x0000000000600e20 is .fini_array
    0x0000000000600e20 - 0x0000000000600e28 is .jcr
    0x0000000000600e28 - 0x0000000000600ff8 is .dynamic
    0x0000000000600ff8 - 0x0000000000601000 is .got
    0x0000000000601000 - 0x0000000000601028 is .got.plt
    0x0000000000601028 - 0x0000000000601034 is .data
    0x0000000000601034 - 0x0000000000601038 is .bss
```

注意 `Entry point: 0x400430` 这一行。现在我们知道我们程序入口点的真正地址。让我们在这个地址下一个断点，然后运行我们的程序，看看会发生什么：

```
(gdb) break *0x400430
Breakpoint 1 at 0x400430
(gdb) run
Starting program: /home/alex/program

Breakpoint 1, 0x0000000000400430 in _start ()
```

有趣。我们并没有看见 `main` 函数的执行，但是我们看见另外一个函数被调用。这个函数是 `_start` 而且根据调试器展现给我们看的，它是我们程序的真正入口。那么，这个函数是从哪里来的，又是谁调用了这个 `main` 函数，什么时候调用的。我会在后续部分尝试回答这些问题。

内核如何运行新程序

首先，让我们来看一下下面这个简单的 `C` 程序：

```
// program.c

#include <stdlib.h>
#include <stdio.h>

static int x = 1;

int y = 2;

int main(int argc, char *argv[]) {
    int z = 3;

    printf("x + y + z = %d\n", x + y + z);

    return EXIT_SUCCESS;
}
```

我们可以确定这个程序按照我们预期那样工作。让我们来编译它：

```
$ gcc -Wall program.c -o sum
```

并且执行：

```
$ ./sum
x + y + z = 6
```

好的，直到现在所有事情看起来听挺好。你可能已经知道一个特殊的系统调用家族 - exec* 系统调用。正如我们从帮助手册中读到的：

> The exec() family of functions replaces the current process image with a new process image.

如果你已经阅读过系统调用章节的第四部分，你可能就知道 execve 这个系统调用定义在 files/exec.c 文件中，并且如下所示，

```
SYSCALL_DEFINE3(execve,
        const char __user *, filename,
        const char __user *const __user *, argv,
        const char __user *const __user *, envp)
{
    return do_execve(getname(filename), argv, envp);
}
```

它以可执行文件的名字，命令行参数的集合以及环境变量的集合作为参数。正如你猜测的，每一件事都是 `do_execve` 函数完成的。在这里我将不描述这个函数的实现细节，因为你可以从这里读到。但是，简而言之， `do_execve` 函数会检查诸如文件名是否有效，未超出进程数目限制等等。在这些检查之后，这个函数会解析 ELF 格式的可执行文件，为新的可执行文件创建内存描述符，并且在栈，堆等内存区域填上适当的值。当二进制镜像设置完成， `start_thread` 函数会设置一个新的进程。这个函数是框架相关的，而且对于 x86_64 框架，它的定义是在 arch/x86/kernel/process_64.c 文件中。

`start_thread` 为段寄存器设置新的值。从这一点开始，新进程已经准备就绪。一旦进程切换)完成，控制权就会返回到用户空间，并且新的可执行文件将会执行。

这就是所有内核方面的内容。Linux 内核为执行准备二进制镜像，而且它的执行从上下文切换开始，结束之后将控制权返回用户空间。但是它并不能回答像 `_start` 来自哪里这样的问题。让我们在下一段尝试回答这些问题。

用户空间程序如何启动

在之前的段落汇总，我们看到了内核是如何为可执行文件运行做准备工作的。让我们从用户空间来看这相同的工作。我们已经知道一个程序的入口点是 `_start` 函数。但是这个函数是从哪里来的呢？它可能来自于一个库。但是如果你记得清楚的话，我们在程序编译过程中并没有链接任何库。

```
$ gcc -Wall program.c -o sum
```

你可能会猜 `_start` 来自于标准库。是的，确实是这样。如果你尝试去重新编译我们的程序，并给 gcc 传递可以开启 `verbose mode` 的 `-v` 选项，你会看到下面的长输出。我们并不对整体输出感兴趣，让我们来看一下下面的步骤：

首先，使用 `gcc` 编译我们的程序：

```
$ gcc -v -ggdb program.c -o sum
...
...
...
/usr/libexec/gcc/x86_64-redhat-linux/6.1.1/cc1 -quiet -v program.c -quiet -dumpbase pr
ogram.c -mtune=generic -march=x86-64 -auxbase test -ggdb -version -o /tmp/ccvUWZkF.s
...
...
...
```

`cc1` 编译器将编译我们的 `C` 代码并且生成 `/tmp/ccvUWZkF.s` 汇编文件。之后我们可以看见我们的汇编文件被 `GNU as` 编译器编译为目标文件：

```
$ gcc -v -ggdb program.c -o sum
...
...
...
as -v --64 -o /tmp/cc79wZSU.o /tmp/ccvUWZkF.s
...
...
...
```

最后我们的目标文件会被 `collect2` 链接到一起：

```
$ gcc -v -ggdb program.c -o sum
...
...
...
/usr/libexec/gcc/x86_64-redhat-linux/6.1.1/collect2 -plugin /usr/libexec/gcc/x86_64-re
dhat-linux/6.1.1/liblto_plugin.so -plugin-opt=/usr/libexec/gcc/x86_64-redhat-linux/6.1
.1/lto-wrapper -plugin-opt=-fresolution=/tmp/ccLEGYra.res -plugin-opt=-pass-through=-l
gcc -plugin-opt=-pass-through=-lgcc_s -plugin-opt=-pass-through=-lc -plugin-opt=-pass-
through=-lgcc -plugin-opt=-pass-through=-lgcc_s --build-id --no-add-needed --eh-frame-
hdr --hash-style=gnu -m elf_x86_64 -dynamic-linker /lib64/ld-linux-x86-64.so.2 -o test
 /usr/lib/gcc/x86_64-redhat-linux/6.1.1/../../../../lib64/crt1.o /usr/lib/gcc/x86_64-r
edhat-linux/6.1.1/../../../../lib64/crti.o /usr/lib/gcc/x86_64-redhat-linux/6.1.1/crtb
egin.o -L/usr/lib/gcc/x86_64-redhat-linux/6.1.1 -L/usr/lib/gcc/x86_64-redhat-linux/6.1
.1/../../../../lib64 -L/lib/../lib64 -L/usr/lib/../lib64 -L. -L/usr/lib/gcc/x86_64-red
hat-linux/6.1.1/../../.. /tmp/cc79wZSU.o -lgcc --as-needed -lgcc_s --no-as-needed -lc
-lgcc --as-needed -lgcc_s --no-as-needed /usr/lib/gcc/x86_64-redhat-linux/6.1.1/crtend
.o /usr/lib/gcc/x86_64-redhat-linux/6.1.1/../../../../lib64/crtn.o
...
...
...
```

是的，我们可以看见一个很长的命令行选项列表被传递给链接器。让我们从另一条路行进。我们知道我们的程序都依赖标准库。

```
$ ldd program
    linux-vdso.so.1 (0x00007ffc9afd2000)
    libc.so.6 => /lib64/libc.so.6 (0x00007f56b389b000)
    /lib64/ld-linux-x86-64.so.2 (0x0000556198231000)
```

从那里我们会用一些库函数，像 `printf` 。但是不止如此。这就是为什么当我们给编译器传递 `-nostdlib` 参数，我们会收到错误报告：

```
$ gcc -nostdlib program.c -o program
/usr/bin/ld: warning: cannot find entry symbol _start; defaulting to 000000000040017c
/tmp/cc02msGW.o: In function `main':
/home/alex/program.c:11: undefined reference to `printf'
collect2: error: ld returned 1 exit status
```

除了这些错误，我们还看见 `_start` 符号未定义。所以现在我们可以确定 `_start` 函数来自于标准库。但是即使我们链接标准库，它也无法成功编译：

```
$ gcc -nostdlib -lc -ggdb program.c -o program
/usr/bin/ld: warning: cannot find entry symbol _start; defaulting to 0000000000400350
```

好的，当我们使用 `/usr/lib64/libc.so.6` 链接我们的程序，编译器并不报告标准库函数的未定义引用，但是 `_start` 符号仍然未被解析。让我们重新回到 `gcc` 的冗长输出，看看 `collect2` 的参数。我们现在最重要的问题是我们的程序不仅链接了标准库，还有一些目标文件。第一个目标文件是 `/lib64/crt1.o` 。而且，如果我们使用 `objdump` 工具去看这个目标文件的内部，我们将看见 `_start` 符号：

```
$ objdump -d /lib64/crt1.o

/lib64/crt1.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <_start>:
   0:	31 ed                	xor    %ebp,%ebp
   2:	49 89 d1             	mov    %rdx,%r9
   5:	5e                   	pop    %rsi
   6:	48 89 e2             	mov    %rsp,%rdx
   9:	48 83 e4 f0          	and    $0xfffffffffffffff0,%rsp
   d:	50                   	push   %rax
   e:	54                   	push   %rsp
   f:	49 c7 c0 00 00 00 00 	mov    $0x0,%r8
  16:	48 c7 c1 00 00 00 00 	mov    $0x0,%rcx
  1d:	48 c7 c7 00 00 00 00 	mov    $0x0,%rdi
  24:	e8 00 00 00 00       	callq  29 <_start+0x29>
  29:	f4                   	hlt
```

因为 `crt1.o` 是一个共享目标文件，所以我们只看到桩而不是真正的函数调用。让我们来看一下 `_start` 函数的源码。因为这个函数是框架相关的，所以 `_start` 的实现是在 sysdeps/x86_64/start.S 这个汇编文件中。

`_start` 始于对 `ebp` 寄存器的清零，正如 ABI) 所建议的。

```
xorl %ebp, %ebp
```

之后，将终止函数的地址放到 `r9` 寄存器中：

```
mov %RDX_LP, %R9_LP
```

正如 ELF 标准所述，

> After the dynamic linker has built the process image and performed the relocations, each shared object gets the opportunity to execute some initialization code. ... Similarly, shared objects may have termination functions, which are executed with the atexit (BA_OS) mechanism after the base process begins its termination sequence.

所以我们需要把终止函数的地址放到 `r9` 寄存器，因为将来它会被当作第六个参数传递给 `__libc_start_main` 。注意，终止函数的地址初始是存储在 `rdx` 寄存器。除了 `%rdx` 和 `%rsp` 之外的其他寄存器保存未确定的值。 `_start` 函数中真正的重点是调用 `__libc_start_main` 。所以下一步就是为调用这个函数做准备。

`__libc_start_main` 的实现是在 csu/libc-start.c 文件中。让我们来看一下这个函数：

```
STATIC int LIBC_START_MAIN (int (*main) (int, char **, char **),
                             int argc,
                            char **argv,
                             __typeof (main) init,
                            void (*fini) (void),
                            void (*rtld_fini) (void),
                            void *stack_end)
```

It takes address of the `main` function of a program, `argc` and `argv`. `init` and `fini` functions are constructor and destructor of the program. The `rtld_fini` is termination function which will be called after the program will be exited to terminate and free dynamic section. The last parameter of the `__libc_start_main` is the pointer to the stack of the program. Before we can call the `__libc_start_main` function, all of these parameters must be prepared and passed to it. Let's return to the sysdeps/x86_64/start.S assembly file and continue to see what happens before the `__libc_start_main` function will be called from there.

该函数以程序 `main` 函数的地址，`argc` 和 `argv` 作为输入。`init` 和 `fini` 函数分别是程序的构造函数和析构函数。`rtld_fini` 是当程序退出时调用的终止函数，用来终止以及释放动态段。`__libc_start_main` 函数的最后一个参数是一个指向程序栈的指针。在我们调用 `__libc_start_main` 函数之前，所有的参数都要被准备好，并且传递给它。让我们返回 sysdeps/x86_64/start.S 这个文件，继续看在 `__libc_start_main` 被调用之前发生了什么。

我们可以从栈上获取我们所需的 `__libc_start_main` 的所有参数。当 `_start` 被调用的时候，我们的栈如下所示：

```
+-----------------+
|       NULL      |
+-----------------+
|       envp      |
+-----------------+
|       NULL      |
+------------------
|       argv      | <- rsp
+------------------
|       argc      |
+-----------------+
```

当我们清零了 `ebp` 寄存器，并且将终止函数的地址保存到 `r9` 寄存器中之后，我们取出栈顶元素，放到 `rsi` 寄存器中。最终 `rsp` 指向 `argv` 数组，`rsi` 保存传递给程序的命令行参数的数目：

```
+-----------------+
|       NULL      |
+-----------------+
|       envp      |
+-----------------+
|       NULL      |
+------------------
|       argv      | <- rsp
+-----------------+
```

这之后，我们将 `argv` 数组的地址赋值给 `rdx` 寄存器中。

```
popq %rsi
mov %RSP_LP, %RDX_LP
```

从这一时刻开始，我们已经有了 `argc` 和 `argv` 。我们仍要将构造函数和析构函数的指针放到合适的寄存器，以及传递指向栈的指针。下面汇编代码的前三行按照 ABI 中的建议设置栈为 `16` 字节对齐，并将 `rax` 压栈：

```
and  $~15, %RSP_LP
pushq %rax

pushq %rsp
mov $__libc_csu_fini, %R8_LP
mov $__libc_csu_init, %RCX_LP
mov $main, %RDI_LP
```

栈对齐之后，我们压栈栈的地址，并且将构造函数和析构函数的地址放到 `r8` 和 `rcx` 寄存器中，同时将 `main` 函数的地址放到 `rdi` 寄存器中。从这个时刻开始，我们可以调用 csu/libc-start.c 中的 `__libc_start_main` 函数。

在我们查看 `__libc_start_main` 函数之前，让我们添加 `/lib64/crt1.o` 文件并且再次尝试编译我们的程序：

```
$ gcc -nostdlib /lib64/crt1.o -lc -ggdb program.c -o program
/lib64/crt1.o: In function `_start':
(.text+0x12): undefined reference to `__libc_csu_fini'
/lib64/crt1.o: In function `_start':
(.text+0x19): undefined reference to `__libc_csu_init'
collect2: error: ld returned 1 exit status
```

现在我们看见了另外一个错误 - 未找到 `__libc_csu_fini` 和 `__libc_csu_init` 。我们知道这两个函数的地址被传递给 `__libc_start_main` 作为参数，同时这两个函数还是我们程序的构造函数和析构函数。但是在 `C` 程序中，构造函数和析构函数意味着什么呢？我们已经在

ELF 标准中看到：

> After the dynamic linker has built the process image and performed the relocations, each shared object gets the opportunity to execute some initialization code. ... Similarly, shared objects may have termination functions, which are executed with the atexit (BA_OS) mechanism after the base process begins its termination sequence.

所以链接器除了一般的段，如 `.text` , `.data` 之外创建了两个特殊的段：

- `.init`
- `.fini`

We can find it with `readelf` util:

我们可以通过 `readelf` 工具找到它们：

```
$ readelf -e test | grep init
  [11] .init             PROGBITS         00000000004003c8  000003c8

$ readelf -e test | grep fini
  [15] .fini             PROGBITS         0000000000400504  00000504
```

这两个将被替换为二进制镜像的开始和结尾，包含分别被称为构造函数和析构函数的例程。这些例程的要点是在程序的真正代码执行之前，做一些初始化/终结，像全局变量如 errno ，为系统例程分配和释放内存等等。

你可能可以从这些函数的名字推测，这两个会在 `main` 函数之前和之后被调用。 `.init` 和 `.fini` 段的定义在 `/lib64/crti.o` 中。如果我们添加这个目标文件：

```
$ gcc -nostdlib /lib64/crt1.o /lib64/crti.o  -lc -ggdb program.c -o program
```

我们不会收到任何错误报告。但是让我们尝试去运行我们的程序，看看发生什么：

```
$ ./program
Segmentation fault (core dumped)
```

是的，我们收到 `segmentation fault` 。让我们通过 `objdump` 看看 `lib64/crti.o` 的内容：

```
$ objdump -D /lib64/crti.o

/lib64/crti.o:     file format elf64-x86-64


Disassembly of section .init:

0000000000000000 <_init>:
   0:	48 83 ec 08          	sub    $0x8,%rsp
   4:	48 8b 05 00 00 00 00 	mov    0x0(%rip),%rax        # b <_init+0xb>
   b:	48 85 c0             	test   %rax,%rax
   e:	74 05                	je     15 <_init+0x15>
  10:	e8 00 00 00 00       	callq  15 <_init+0x15>

Disassembly of section .fini:

0000000000000000 <_fini>:
   0:	48 83 ec 08          	sub    $0x8,%rsp
```

正如上面所写的， `/lib64/crti.o` 目标文件包含 `.init` 和 `.fini` 段的定义，但是我们可以看见这个函数的桩。让我们看一下 sysdeps/x86_64/crti.S 文件中的源码：

```
    .section .init,"ax",@progbits
    .p2align 2
    .globl _init
    .type _init, @function
_init:
    subq $8, %rsp
    movq PREINIT_FUNCTION@GOTPCREL(%rip), %rax
    testq %rax, %rax
    je .Lno_weak_fn
    call *%rax
.Lno_weak_fn:
    call PREINIT_FUNCTION
```

它包含 `.init` 段的定义，而且汇编代码设置 16 字节的对齐。之后，如果它不是零，我们调用 `PREINIT_FUNCTION` ；否则不调用：

```
00000000004003c8 <_init>:
  4003c8:       48 83 ec 08             sub    $0x8,%rsp
  4003cc:       48 8b 05 25 0c 20 00    mov    0x200c25(%rip),%rax        # 600ff8 <_D
YNAMIC+0x1d0>
  4003d3:       48 85 c0                test   %rax,%rax
  4003d6:       74 05                   je     4003dd <_init+0x15>
  4003d8:       e8 43 00 00 00          callq  400420 <__libc_start_main@plt+0x10>
  4003dd:       48 83 c4 08             add    $0x8,%rsp
  4003e1:       c3                      retq
```

where the `PREINIT_FUNCTION` is the `__gmon_start__` which does setup for profiling. You may note that we have no return instruction in the sysdeps/x86_64/crti.S. Actually that's why we got segmentation fault. Prolog of `_init` and `_fini` is placed in the sysdeps/x86_64/crtn.S assembly file:

其中，`PREINIT_FUNCTION` 是设置简况的 `__gmon_start__`。你可能发现，在 sysdeps/x86_64/crti.S中，我们没有 `return` 指令。事实上，这就是我们获得 `segmentation fault` 的原因。`_init` 和 `_fini` 的序言被放在 sysdeps/x86_64/crtn.S 汇编文件中：

```
.section .init,"ax",@progbits
addq $8, %rsp
ret

.section .fini,"ax",@progbits
addq $8, %rsp
ret
```

如果我们把它加到编译过程中，我们的程序会被成功编译和运行。

```
$ gcc -nostdlib /lib64/crt1.o /lib64/crti.o /lib64/crtn.o  -lc -ggdb program.c -o prog
ram

$ ./program
x + y + z = 6
```

结论

现在让我们回到 `_start` 函数，以及尝试去浏览 `main` 函数被调用之前的完整调用链。

`_start` 总是被默认的 `ld` 脚本链接到程序 `.text` 段的起始位置：

```
$ ld --verbose | grep ENTRY
ENTRY(_start)
```

`_start` 函数定义在 sysdeps/x86_64/start.S 汇编文件中，并且在 `__libc_start_main` 被调用之前做一些准备工作，像从栈上获取 `argc/argv`，栈准备等。来自于 csu/libc-start.c 文件中的 `__libc_start_main` 函数注册构造函数和析构函数，开启线程，做一些安全相关的操作，比如在有需要的情况下设置 `stack canary`，调用初始化，最后调用程序的 `main` 函数以及返回结果退出。而构造函数和析构函数分别是 `main` 之前和之后被调用。

```
result = main (argc, argv, __environ MAIN_AUXVEC_PARAM);
exit (result);
```

结束

链接

- system call
- gdb
- execve
- ELF
- x86_64
- segment registers
- context switch
- System V ABI

Linux 内核内部 `系统` 数据结构

这不是 `linux-insides-zh` 中的一般章节。正如你从题目中理解到的，它主要描述 Linux 内核中的内部 `系统` 数据结构。比如说，中断描述符表 (`Interrupt Descriptor Table`), 全局描述符表 (`Global Descriptor Table`) 。

大部分信息来自于 Intel 和 AMD 官方手册。

中断描述符（IDT）

三个常见的中断和异常来源：

- 异常 - sync；
- 软中断 - sync；
- 外部中断 - async。

异常的类型：

- 故障 - 在指令导致异常 `之前` 会被准确地报告。 `%rip` 保存的指针指向故障的指令；
- 陷阱 - 在指令导致异常 `之后` 会被准确地报告。 `%rip` 保存的指针同样指向故障的指令；
- 终止 - 是不明确的异常。 因为它们不能被明确，中止通常不允许程序可靠地再次启动。

只有当RFLAGS.IF = 1时， `可屏蔽` 中断触发才中断处理程序。 除非RFLAGS.IF位清零，否则它们将持续处于等待处理状态。

`不可屏蔽` 中断（NMI）不受rFLAGS.IF位的影响。 无论怎样一个NMI的发生都会进一步屏蔽之后的其他NMI，直到执行IRET（中断返回）指令。

具体的异常和中断来源被分配了固定的向量标识号（也称“中断向量”或简称“向量”）。中断处理程序使用中断向量来定位异常或中断，从而分配相应的系统软件服务处理程序。有至多256个特殊的中断向量可用。前32个是保留的，用于预定义的异常和中断条件。请参考arch / x86 / include / asm / traps.h头文件中对他们的定义：

```
/* 中断/异常 */
enum {
    X86_TRAP_DE = 0,    /*  0, 除零错误 */
    X86_TRAP_DB,        /*  1, 调试 */
    X86_TRAP_NMI,        /*  2, 不可屏蔽中断 */
    X86_TRAP_BP,        /*  3, 断点 */
    X86_TRAP_OF,        /*  4, 溢出 */
    X86_TRAP_BR,        /*  5, 超出范围 */
    X86_TRAP_UD,        /*  6, 操作码无效 */
    X86_TRAP_NM,        /*  7, 设备不可用 */
    X86_TRAP_DF,        /*  8, 双精度浮点错误 */
    X86_TRAP_OLD_MF,    /*  9, 协处理器段溢出 */
    X86_TRAP_TS,        /* 10, 无效的 TSS */
    X86_TRAP_NP,        /* 11, 段不存在 */
    X86_TRAP_SS,        /* 12, 堆栈段故障 */
    X86_TRAP_GP,        /* 13, 一般保护故障 */
    X86_TRAP_PF,        /* 14, 页错误 */
    X86_TRAP_SPURIOUS,    /* 15, 伪中断 */
    X86_TRAP_MF,        /* 16, x87 浮点异常 */
    X86_TRAP_AC,        /* 17, 对齐检查 */
    X86_TRAP_MC,        /* 18, 机器检测 */
    X86_TRAP_XF,        /* 19, SIMD （单指令多数据结构浮点）异常 */
    X86_TRAP_IRET = 32,    /* 32, IRET （中断返回）异常 */
};
```

错误代码（Error code）

处理器异常处理程序使用错误代码报告某些异常的错误和状态信息。在控制权交给异常处理程序期间，异常处理装置将错误代码推送到堆栈中。错误代码有两种格式：

- 多数异常错误报告格式；
- 页错误格式。

选择子错误代码的格式如下：

```
31                           16 15                                  3   2   1   0
+-------------------------------------------------------------------------------+
|                              |                                    | T | I | E |
|           Reserved           |             Selector Index         | - | D | X |
|                              |                                    | I | T | T |
+-------------------------------------------------------------------------------+
```

说明如下:

- `EXT` - 如果该位设置为1，则异常源在处理器外部。 如果设置为0，则异常源位于处理器的内部；
- `IDT` - 如果该位设置为1，则错误代码选择子索引字段引用位于“中断描述符表”中的门描

述符。 如果设置为0，则选择子索引字段引用“全局描述符表”或本地描述符表“LDT”中的描述符，由“TI”位所指示；

- `TI` - 如果该位设置为1，则错误代码选择子索引字段引用“LDT”中的描述符。 如果清除为0，则选择子索引字段引用“GDT”中的描述符；
- `Selector Index` - 选择子索引字段指定索引为“GDT‘，“LDT”或“IDT”，它是由“IDT”和“TI”位指定的。

页错误代码格式如下：

```
31                                                              4   3   2   1   0
+-------------------------------------------------------------------------------+
|                                                  |     | R | U | R | - |
|                      Reserved                    | I/D | S | - | - | P |
|                                                  |     | V | S | W | - |
+-------------------------------------------------------------------------------+
```

说明如下:

- `I/D` - 如果该位设置为1，表示造成页错误的访问是取指；
- `RSV` - 如果该位设置为1，则页错误是处理器从保留给分页表的区域中读取1的结果;
- `U/S` - 如果该位被设置为0，则是管理员模式（`CPL = 0,1或2`）进行访问导致了页错误。 如果该位设置为1，则是用户模式（CPL = 3）进行访问导致了页错误；
- `R/W` - 如果该位被设置为0，导致页错误的是内存读取。 如果该位设置为1，则导致页错误的是内存写入；
- `P` - 如果该位被设置为0，则页错误是由不存在的页面引起的。 如果该位设置为1，页错误是由于违反页保护引起的。

中断控制传输（Interrupt Control Transfers）

IDT可以包含三种门描述符中的任何一种：

- `Task Gate（任务门）` - 包含用于异常与或中断处理程序任务的TSS的段选择子；
- `Interrupt Gate（中断门）` - 包含处理器用于将程序从执行转移到中断处理程序的段选择子和偏移量；
- `Trap Gate（陷阱门）` - 包含处理器用于将程序从执行转移到异常处理程序的段选择子和偏移量。

门的一般格式是：

```
127                                                                             96
+-------------------------------------------------------------------------------+
|                                                                               |
|                                Reserved                                       |
|                                                                               |
+--------------------------------------------------------------------------------
95                                                                              64
+-------------------------------------------------------------------------------+
|                                                                               |
|                               Offset 63..32                                   |
|                                                                               |
+-------------------------------------------------------------------------------+
63                               48 47       46  44   42    39             34    32
+-------------------------------------------------------------------------------+
|                                  |       |  D  |   |     |      |   |   |     |
|       Offset 31..16              |   P   |  P  | 0 |Type |0 0 0 | 0 | 0 | IST |
|                                  |       |  L  |   |     |      |   |   |     |
 -------------------------------------------------------------------------------+
31                                   16 15                                      0
+-------------------------------------------------------------------------------+
|                                      |                                        |
|          Segment Selector            |                 Offset 15..0           |
|                                      |                                        |
+-------------------------------------------------------------------------------+
```

说明如下：

- `Selector` - 目标代码段的段选择子；
- `Offset` - 处理程序入口点的偏移量；
- `DPL` - 描述符权限级别；
- `P` - 当前段标志；
- `IST` - 中断堆栈表；
- `TYPE` - 本地描述符表（LDT）段描述符，任务状态段（TSS）描述符，调用门描述符，中断门描述符，陷阱门描述符或任务门描述符之一。

`IDT` 描述符在Linux内核中由以下结构表示（仅适用于 `x86_64`）：

```
struct gate_struct64 {
    u16 offset_low;
    u16 segment;
    unsigned ist : 3, zero0 : 5, type : 5, dpl : 2, p : 1;
    u16 offset_middle;
    u32 offset_high;
    u32 zero1;
} __attribute__((packed));
```

它定义在 arch/x86/include/asm/desc_defs.h 头文件中。

任务门描述符不包含 `IST` 字段，并且其格式与中断/陷阱门不同：

```
struct ldttss_desc64 {
    u16 limit0;
    u16 base0;
    unsigned base1 : 8, type : 5, dpl : 2, p : 1;
    unsigned limit1 : 4, zero0 : 3, g : 1, base2 : 8;
    u32 base3;
    u32 zero1;
} __attribute__((packed));
```

任务切换期间的异常（Exceptions During a Task Switch）

任务切换在加载段选择子期间可能会发生异常。页错误也可能会在访问TSS时出现。在这些情况下，由硬件任务切换机构完成从TSS加载新的任务状态，然后触发适当的异常处理。

在长模式下，由于硬件任务切换机构被禁用，因而在任务切换期间不会发生异常。

不可屏蔽中断（Nonmaskable interrupt）

未完待续

API

未完待续

中断堆栈表（Interrupt Stack Table）

未完待续

有帮助的链接

Linux 启动

- Linux/x86 boot protocol
- Linux kernel parameters

保护模式

- 64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf

串口编程

- 8250 UART Programming
- Serial ports on OSDEV

VGA

- Video Graphics Array (VGA)

IO

- IO port programming

GCC and GAS

- GCC type attributes
- Assembler Directives

重要的数据结构

- task_struct definition

其他框架

- PowerPC and Linux Kernel Inside

有帮助的链接

- Linux x86 Program Start Up
- Memory Layout in Program Execution (32 bits)

翻译人员 (排名不分先后)

@xinqiu

@lijiangsheng1

@littleneko

@qianmoke

@icecoobe

@choleraehyq

@mudongliang

@oska874

@cloudusers

@hailincai

@zmj1316

@zhangyangjing

@huxq

@worldwar

@keltoy

@a1ickgu0

@hao-lee

@woodpenker

@tjm-1990

@up2wing

@NeoCui